CONTENTS

1 Introduction 3

2 Quick Start Guide 5
  2.1 Installation ................................................................. 5
  2.2 Running the Pipeline .................................................... 5

3 Overview of Software Structure 9
  3.1 File Layout ................................................................. 9
  3.2 Flowcharts ................................................................. 10

4 Practical Examples 13
  4.1 Modern text ................................................................. 13
  4.2 Historical documents (relating to England) .......................... 16
  4.3 Classical texts .............................................................. 18
  4.4 Using pre-formatted input .............................................. 18

5 The Pipeline 21
  5.1 Geotagging ................................................................. 21
  5.2 Georesolution .............................................................. 53

6 Gazetteers 59
  6.1 Online Resources ......................................................... 59
  6.2 Options for Local Gazetteers .......................................... 60

7 The Edina Unlock Service 63
  7.1 Unlock Places .............................................................. 63
  7.2 Unlock Text ................................................................. 63

8 Appendix 1: LT-TTT2 Tutorial 65

9 Appendix 2: LTG Publications about the Geoparser 67
CHAPTER

ONE

INTRODUCTION

The Edinburgh Geoparser is a language processing tool designed to detect placename references in English text and ground them against an authoritative gazetteer so that they can be plotted on a map. The two main processes involved are entity recognition, to find the placename mentions and categorise them as such, followed by a ranking process that selects the likeliest location for each place from what may be a long list of candidates.

The Quick Start Guide explains how to install the software and start using it, and there are some worked examples of how to use it, with illustrations of the output produced, in the Practical Examples chapter.

The geoparser was developed by Claire Grover and Richard Tobin, of the Language Technology Group (LTG) in the School of Informatics at Edinburgh University. Over a number of years they and other colleagues from the LTG have refined and added to the geoparser’s functionality. Appendix 2: LTG Publications about the Geoparser contains a list of some published papers evaluating the geoparser’s performance relative to other similar systems, and discussing how it has been used by the LTG and our partners in various projects.

Like many linguistic tools of this kind, the geoparser software is designed to work in a “pipeline”, where the output of one process forms the input for the next. This construction gives flexibility and makes it relatively easy to switch components in and out - so if you prefer your own tokeniser to ours, say, it is easy to make the substitution. The Pipeline chapter explains the two steps, geotagging to find the placenames, and georesolution to ground them in space. See the Geotagging section for details on changing the linguistic components. The Overview of Software Structure chapter contains flowcharts and diagrams of how the whole pipeline fits together.

The geoparser is configured to work with a number of different gazetteers, as explained in the Gazetteers chapter. Although primarily designed to detect and geo-locate spatial references, the pipeline has evolved to find and categorise other entity categories, viz person, organisation and time expressions, as well as location. A range of visualisation files can be produced, including a display that shows all entity categories plus temporal events detected.

The geoparser works best with fairly short texts (up to a few pages), for reasons that are explained in the Geo-resolution section. Therefore if you have a very large corpus to process, it’s advisable to divide it into smaller chunks.

This documentation covers the downloadable version of the Edinburgh Geoparser, to be installed on your own local machine. There is also an online version embedded in the Edina Unlock Text service, which is described in the Unlock chapter.

We expect the geoparser to continue to evolve, and already have plans for enhancements. We welcome suggestions and collaboration, so please get in touch if you have ideas about how we should develop the software.
CHAPTER TWO

QUICK START GUIDE

2.1 Installation

To install the Edinburgh Geoparser, download the software bundle from the LTG’s geoparser software page and unpack it in a suitable location (in your home directory, say). The directory structure produced will be as shown in the File layout Figure.

The visualisation components use Google Maps and the gazmap and gazmap-top scripts contain API keys obtained for the ed.ac.uk domain, held in the defkey variable. That kind of API key is no longer available from Google so, rather than suggest you insert your own key, we have left ours in place. If you do have a suitable API key (obtained before 2013) please insert it in these scripts in place of ours.

The geoparser runs on linux and Macintosh platforms, both 32 and 64 bit. The underlying LT-XML2 components are available in source code for local compilation, from the LTG software page, but some required components are binary only.

The geoparser can reference a range of different gazetteers, hosted on the web, on Edina’s Unlock service or locally. For the web-based and Unlock gazetteers (see -g gazetteer parameter) no additional software is needed.

It is possible that you will want to set up a local copy of a gazetteer and in this case you will obviously need to install and manage it. The pros and cons of using a local gazetteer are discussed in the Options for Local Gazetteers section and two examples - for which the geoparser is already configured - are described: Geonames and Pleiades. Both of these examples use a locally managed MySQL database. If you plan to use the geonames-local or plplus-local options you will need to set up the gazetteers as described and edit the gazlookup-geonames-local and gazlookup-plplus-local scripts to contain the correct connection string for your MySQL database. This is also explained in the Options for Local Gazetteers section.

2.2 Running the Pipeline

To test the pipeline, do this:

```bash
cd scripts
cat ../in/172172.txt | ./run -t plain -g unlock
```

This uses the option of plain text input and uses unlock as the gazetteer. The output xml file is sent to stdout.

Note that the order of the -t and -g options is immaterial. This applies to all the command line options.

2.2.1 Visualisation output: -o

To run and create visualisation files:

```bash
cat ../in/172172.txt | ./run -t plain -g unlock -o ../out 172172
```

Same as before except that -o takes two args, an output directory ../out and a prefix for the output file names 172172. The output directory must already exist. The results appear in the output directory ../out:
• 172172.display.html is the geoparser map display.
• 172172.timeline.html is the timeline display (note that person, location, organisation and date entities are highlighted in this display).
• 172172.out.xml is the output that goes to stdout when it is run without --o.

The other files are ones used or the map and timeline display or ones which may be useful in their own right.

2.2.2 Single placename markers: -top

By default, all candidate placenames are shown in the display, with the top-ranked one in green and the rest in red. If the -top option is added to the command line then three extra display files will be created, which show only the top-ranked candidate for each place, not all the alternatives considered. For the example used above the extra files would be:

• 172172.gazlist-top.html is the geoparser map display with only one placename marker per toponym.

2.2.3 Input type and gazetteer: -t -g

The options for -t type and -g gazetteer are:

- t plain (plain text)
  ltgxml (xml file in a certain format with paragraphs marked up)
  gb (Google Books html files)
- g unlock (Edina’s gazetteer of mainly UK placenames)
  os (Just the OS part of Unlock)
  naturalearth (Just the Natural Earth part of Unlock)
  geonames (online world-wide gazetteer)
  plplus (Pleiades+ gazetteer of ancient places, on Edina)
  deep (DEEP gazetteer of historical placenames in England)

[ geonames-local (locally maintained copy on ed.ac.uk network) ]
[ plplus-local (locally maintained Pleiades+, with geonames lookup) ]

The last two gazetteer options will only be usable if local gazetteers are maintained; they are included in case useful. See Options for Local Gazetteers for how to make use of them.

If your input is xml with paragraphs already marked, it may be worth converting it to ltgxml format. See the example in/172172.xml for the format.

For Google Books input, which can be extremely untidy, pre-processing is done to ensure it doesn’t break the xml processes in the pipeline.

2.2.4 Docdate: -d

If you know the creation/writing date of the document you can supply this with -d docdate:

cat ../in/172172.txt | ./run -t plain -g unlock -d 2010-08-13

This will be used in event and relation detection and timeline display.

2.2.5 Limiting geographical area: -l -lb

If you know that toponyms in your text are likely to be in a particular geographical area you can specify a bounding circle -l locality or a rectangular -lb locality box. The geoparser will prefer places in the area specified but will still choose locations outside it if other factors give them higher weighting.

To specify a circular locality:

-\( l \) \( lat \) long radius score

where

-\( l \)at and long are in decimal degrees (ie 57.5 for 57 degrees 30 mins)
- radius is in km
- score is a numeric weight assigned to locations within the area (else 0).

To specify a locality box:

-\( l\)b W N E S score

where

- W(est) N(orth) E(ast) S(outh) are decimal degrees
- score is as for -l option.

2.2.6 DEEP only options: -c -r

For DEEP a new -\( c \) county option has been added. This allows the user to specify the county that the document is about in order to only consider DEEP gaz entries for that county. Multiple uses of -\( c \) allow several counties to be specified. For example:

```
cat <infile> | ./run -t plain -g deep -c Oxfordshire -c Wiltshire
```

The values for -\( c \) are the county names in the DEEP gazetteer:


Note that county names with white space need to be enclosed in double quotes:

```
cat <infile> | ./run -t plain -g deep -c Oxfordshire -c Wiltshire -c "North Riding of Yorkshire" -c "East Riding of Yorkshire" -c "West Riding of Yorkshire"
```

A new -\( r \) begindate enddate option is also available for DEEP to restrict the choice of DEEP gazetteer records which have attestation dates within the date range:

```
cat ../in/essexff.txt | ./run -t plain -g deep -c Essex -r 1000 1400
```

---

1 DEEP, Digital Exposure of English Placenames, was a JISC-funded project to digitise and make available the 86 volumes of the Survey of English Place-Names. See placenames.org.uk for the source material it worked with, which covers the evolution of placenames in England. The 86-volume county by county survey details over four million variant forms, from classical sources, through the Anglo-Saxon period and into medieval England and beyond to the modern period.
CHAPTER THREE

OVERVIEW OF SOFTWARE STRUCTURE

See The Pipeline for a description of the logical structure of the geoparser pipeline, and how to customise it if required. This chapter explains the physical layout of the software directories and provides flowcharts of the run script that drives the pipeline.

3.1 File Layout

The directory structure is as shown in Figure File layout. The scripts directory contains all the driving scripts, with run being the master that will run the entire pipeline. The Flowcharts diagrams show how the subsidiary scripts slot in; these in turn call routines from the lib directory libraries.

There is a setup script called by all other scripts to check the platform and set paths correctly. This checks for an environment variable $LXDEBUG which, if set, puts the pipeline into debug mode, so that intermediate temporary files (in /tmp) are kept for examination instead of being cleaned up when the pipeline exits.
3.2 Flowcharts

This section contains two flowcharts, for the geotagging and geoparsing steps, explaining the physical structure of the software. These may be handy for reference if you are planning to customise the geoparser. They cover the run script that drives the pipeline and calls other scripts in the scripts directory.

The pipeline is designed to be modular so that you can slot in your own components if desired. These flowcharts show the input and output required at each stage. The command line options tested in the decision diamonds are explained in Section Running the Pipeline.

The first chart, Figure Geotagging flowchart, shows the first stage of the pipeline, up to the production of geotagged text output, ie a file with linguistic markup (paragraphs, sentences, tokens, part of speech tags, lemmas) and with Named Entities identified and categorised. The pipeline annotates the input with more than just geographic entities. Personal names, organisations and time expressions are also tagged, along with event relations that can be plotted on a timeline.

The second chart, Figure Georesolution flowchart, covers the second stage, taking the output from step one as input. The pathway will depend on the parameters specified to the run command. Without the -o option, specifying output files destination, the visualisation steps are skipped altogether and the geogrounded textual output goes to standard out. If -o is specified then various display files are created, primarily for mapping (using Google Maps), but including event detection displayed with a Timeline widget and highlighting other entity categories besides location.
3.2. Flowcharts

Figure 3.2: Geotagging flowchart
Chapter 3. Overview of Software Structure

Figure 3.3: Georesolution flowchart
This chapter describes some examples of using the geoparser with text from different domains, such as modern text, historical documents and classical works in English translation. Using the command line parameters you can switch different lexicons and gazetteers on or off, to suit different kinds of text. Each of the examples below looks at different aspects of the output produced.

The examples here are for the domains we have tested, and the relevant files are included in the distribution (in the in directory) so you can run the examples as described below. These are real texts we have worked with, not prepared examples, and the output will contain errors - of precision or recall over the entities, or through mis-identification of locations. The publications in Appendix 2: LTG Publications about the Geoparser discuss the performance you can expect in various domains.

If your domain fits one of these categories you should be able to use the geoparser without adaptation, by simply specifying the \texttt{-t type} and \texttt{-g gazetteer} parameters appropriately. See \texttt{-t and -g parameters} for the available options.

For a discussion of the issues involved in customising the geoparser for a new domain, see \textit{Adapting the Edinburgh Geoparser for Historical Geo-referencing} in the Appendix 2: LTG Publications about the Geoparser chapter.

4.1 Modern text

\textbf{Plain text: “burtons.txt”}

We start with a simple example using the file “burtons.txt”, without creating any visualisation files, and writing to stdout. Here the command is being run from the geoparser root directory, but it could be run from anywhere, with appropriately specified paths:

\texttt{cat in/burtons.txt \mid scripts/run -t plain -g unlock}

The following command, using input redirection instead of a pipe, is of course completely equivalent:

\texttt{scripts/run -t plain -g unlock \mid in/burtons.txt}

This run uses Edina’s Unlock gazetteer which is mainly UK oriented. The input file starts like this:

\texttt{How the home of Mini Rolls and Smash was gobbled up}

\texttt{Food factory workers facing the the sack will march on Saturday for an}
\texttt{economy that values more than just money}

\texttt{Among the thousands of people who join the big anti-cuts march this}
\texttt{Saturday will be a coach load from Wirral. ...}

The output starts like this:

\texttt{<?xml version="1.0" encoding="UTF-8"?>}
\texttt{<document version="3">}
\texttt{<meta>}
\texttt{<attr name="docdate" id="docdate" year="2014" month="07" date="02"}
How the home of Mini Rolls and Smash was gobbled up.

The complete file is here (html documentation only).

The output is xml with paragraphs and sentences marked and individual tokens in <w> elements, with various linguistic attributes added. The unique “id” attribute is based on character position in the input text. Some meta data has been added, including a “docdate” which defaults to the current date as no -d docdate parameter was specified. Placename mentions found in the text will have a “locname” attribute on the <w> element, but this is part of the intermediate processing, and the final Named Entity markup is specified using standoff xml as described below.

The <text> element is followed by a <standoff> section. The following sample shows the structure:

```xml
<sdate="2014-07-02" day-number="735415" day="Wednesday" wdaynum="3">20140702</sdate>
</attr>
<meta>
<text>
<p>
<s id="s1">
<w pws="yes" id="w13" p="WRB" group="B-ADVP">How</w>
<w pws="yes" id="w17" p="DT" group="B-NP">the</w>
<w l="home" pws="yes" id="w21" p="NN" headn="yes" group="I-NP">home</w>
<w pws="yes" id="w26" p="IN" group="B-PP">of</w>
<w common="true" l="minus" pws="yes" id="w29" p="NNP" event="true">
  headn="yes" group="B-NP">Mini</w>
<w common="true" vstem="roll" l="roll" pws="yes" id="w34" p="NNP" event="true">
  headn="yes" group="I-NP">Rolls</w>
<w pws="yes" id="w40" p="CC" headn="yes" group="I-NP">and</w>
<w common="true" l="smash" pws="yes" id="w44" p="NNP" event="true">
  headn="yes" group="I-NP">Smash</w>
<w l="be" pws="yes" id="w50" p="VBD" group="B-VP">was</w>
<w l="gobble" pws="yes" id="w54" p="VBN" headv="yes" group="I-VP">gobbled</w>
<w pws="yes" id="w62" p="RP" group="I-VP">up</w>
</s>
</p>
...
</text>
</meta>

<standoff>
<ents source="ner-rb">
  <ent date="05" month="07" year="2014" sdate="2014-07-05" day-number="735418">
    <parts>
      <part ew="w125" sw="w125">Saturday</part>
    </parts>
  </ent>
</ents>

<ents source="events">
  <ent tense="past" voice="pass" asp="simple" modal="nc" id="ev1" subtype="gobble" type="event">
    <parts>
      <part ew="w54" sw="w54">gobbled</part>
    </parts>
  </ent>
</ents>

(relations source="temprel">
  <relation id="rbr1" type="beforeorincl" text="was gobbled up">
    <argument arg1="true" ref="ev1"/>
    <argument arg2="true" ref="docdate"/>
  </relation>
</relations>
</standoff>
```
There are two sets of `<ents>` elements because the pipeline uses two separate steps. The first is a rule-based process ("ner-rb") to identify and classify the entity mentions - the above example shows a date entity. The entity categories detected are: date, location, person, and organisation. The entities are tied back to their positions in the text by the `<part>` element, which has “sw” (start word) and “ew” (end word) attributes whose values match the “id”s on the `<w>`s in the text.

The second set of `<ents>` are mainly verbs and verb phrases, tagged as a basis for detecting events mentioned in the text. The `<relations>` section relates pairs of `<ent>`s, identified by a “ref” attribute that points to event `<ent>`s (such as “ev1”) or rule-based ones (eg “rb1”) or to the `docdate` as in this example.

From a purely geoparsing point of view, only the rule-based “location” entities may be required, which look like this:

```
<ents source="ner-rb">
  <ent id="rb3" type="location" lat="53.37616" long="-3.10501"
       gazref="geonames:7733088" in-country="GB" feat-type="ppl">
    <parts>
      <part ew="w288" sw="w288">Wirral</part>
    </parts>
  </ent>
</ents>
```

These can easily be extracted if desired. (For example one could extract these with lxgrep or remove other unwanted nodes with lxreplace, both of which are included in the LT-XML2 toolkit). Tools to create the rest of the markup have been added to the pipeline at various times for different projects and the full output is included because why wouldn’t we?

**News text with known date: “172172”**

With news text the date of the story is often known, and can be specified to the geoparser to help with event detection. The next example also specifies the `-o outdir prefix` option so that a full set of visualisation files will be produced in addition to the main output described above (which will be in a file named “outdir/prefix.out.xml”):

```
cat in/172172.txt |
scripts/run -t plain -g geonames -d 2010-08-13 -o out 172172
```

In this case we have directed output to the pipeline’s out directory but it can be sent anywhere using a relative or absolute path. The online Geonames gazetteer has been chosen, as the text doesn’t relate to the UK. It begins:

Nadal and Murray set up semi showdown
(CNN) -- Rafael Nadal and Andy Murray are both through to the semifinals of the Rogers Cup in Toronto, where they will face each other for a place in Sunday’s final.
Murray played some superb tennis in crushing the in-form David Nalbandian but Nadal had to recover from dropping the opening set to get past Germany’s Philipp Kohlschreiber.
Nalbandian won the ATP title in Washington last weekend and came into Friday’s encounter on an 11-match unbeaten streak. ...

Specifying the `-o` option means that, instead of just the tagged text file, we get a collection of output files:

- 172172.display.html
- 172172.events.xml
- 172172.gaz.xml
- 172172.gazlist.html
- 172172.gazmap.html
- 172172.geotagged.html
- 172172.nertagged.xml
- 172172.out.xml
- 172172.timeline.html
The output files are described in the Quick Start Guide, Visualisation output: -o. We looked at the format of the “172172.out.xml” file above. The other main file is “172172.display.html”, which looks as shown in Figure Geoparser display file for news text input. The map window uses Google Maps to display the placename locations, with green markers for the top-ranked candidate for each place and red markers for the other candidates. The bottom left panel shows the input text, with placenames highlighted, and the bottom right panel lists the placenames with all the candidate locations found for each. The first in the list is the chosen one, in green. You can see from the length of the horizontal scroll bar that there are typically a great many other candidates - this is especially true when using Geonames, as common placenames like the ones in this file are repeated many times all over the world. The display is centred on the first placename mention, “Toronto”, and can be re-centred by selecting other lat/long positions from the placename list.

![Image of Geoparser display file](file://...)

**Figure 4.1: Geoparser display file for news text input**

This was quite a short file to try to detect events in, but those found are listed in “172172.events.xml” (available [here](file:///...)) in the html documentation, which is used to produce the Timeline display shown in Figure Timeline file. (Note that the Chrome browser will only display the timeline correctly if served from a web server. Firefox is less fussy and will display the page as illustrated from a file://... URI.)

This display shows other entity categories besides the locations, which are in green. Personal names are in red, organisations in blue and time expressions in yellow. The pipeline detected 5 events in this input but was only able to assign specific dates to two of them, which are the two plotted on the timeline. The other events included references to “this season” and “this year”, which couldn’t be placed on the timeline. In the screen shot, an “info” box has been brought up, by clicking on one of the events. It shows the text of the event and its timestamp.

### 4.2 Historical documents (relating to England)

We now take a more complex example, using some historical text. The input file is “cheshirepleas.txt”, which starts thus:

On Saturday (fn. 2) next after the feast of St. Edward the King in the 33rd year of the reign of King Edward [1305] Robert le Grouynour
Edinburgh Geoparser Documentation, Release 1.0

4.2. Historical documents (relating to England)

Figure 4.2: Timeline file

did his homage and fealty for all the tenements of Lostoke, and acknowledged that he held the manor of Lostoke entirely from the manor of Weverham for homage and service and fealty [fo. 38d (275 d)] and suit at the court of Weverham every fortnight, and 17s. yearly to the manor of Weverham at the four terms, and two customary pigs, and four foot-men in time of war at Chester bridge over the Dee, when Weverham finds eight foot-men, and three when the manor of Weverham finds six, or two when Weverham finds four men, with the ward and relief of Lostok for all service...

The appropriate gazetteer for this text is DEEP, a specialist gazetteer of historical placenames in England (see footnote [1] in the Quick Start Guide for details). If we know that the text is about Cheshire we can restrict the gazetteer to that county. The text deals with dates in the 14th century - in fact over several different years, despite the rather specific sound of “On Saturday next”, so whilst a docdate parameter may not be appropriate, we can limit the DEEP candidates to ones attested for the medieval period, using a date range (say for the 12th to 14th centuries). The run command we will use is:

```
cat in/cheshirepleas.txt | scripts/run -t plain -g deep -c Cheshire -r 1100 1400 -o out chespleas
```

In this case we have specified that a full set of output files should be produced, in the usual out directory and prefixed with the string “chespleas”. Figure Display file for Cheshire input and DEEP gazetteer shows the display file created. The blue underlined placenames in the text window are embedded links back to the source gazetteer material for the toponym at placenames.org.uk.

As expected, the chosen locations are clustered together in Cheshire, the single outlier being a reference to “Wales”.

---

4.2. Historical documents (relating to England)
4.3 Classical texts

As part of the GAP (Google Ancient Places) project, the geoparser was adapted to deal with classical texts in English translation. This requires different lexicons of places and personal names and uses the Pleiades gazetteer of ancient places. (See the Pleiades+ section for details of Pleiades and Pleiades+.)

The geoparser output was post-processed by the GAP project to create the GapVis display (versions 1 and 2 are currently available). This only requires one location per toponym mention so only the top-ranked candidate was passed on. If you only require the “best” location (the green markers in the displays above) then specify the `-top` option:

```
cat in/herodotusBk1.txt | scripts/run -t plain -g plplus -o out hbk1plplus -top
```

The `-top` option can be used for any input and will result in an extra set of display files being produced, in which the unsuccessful candidate locations have been removed. The display page in this example will be file "out/hbk1plplus.display-top.html", which is illustrated in Figure Herodotus display file. The text is the opening of Book 1 of the Histories by Herodotus.

In principle it might be possible to process input in the original Latin or Greek (or indeed in any language), if suitable linguistic components could be substituted in the geotagging stage of the pipeline. This is a project for another day. The Hestia 2 project has taken steps along the way towards allowing students to work with the Greek version of the Histories.

4.4 Using pre-formatted input

The examples above all use plain text input files. If your input files already contain markup, such as html or xml, you may wish to alter the pre-processing steps of the pipeline to cater for it. Alternatively, it may be simpler to strip the markup and treat your input as plain text. The `type` parameter accepts two specific formats, `gb` (Google
4.4.1 Google Books format

Another spin-off from the GAP project was the need to process Google Books input. GAP was a Google-funded project using bulk quantities of Google Books material, specifically classical texts. The original scanning and OCR work was done on a very large scale by Google and the quality can be variable to say the least. The data was made available as html files and we had the choice of either stripping all the markup - which would have thrown away valuable information - or attempting to ingest the raw files. The prepare stage at the start of the pipeline was amended to do just enough pre-processing of the html to ensure that the many non-printable characters contained in the OCR-ed input don’t break the xml tools. Because the files vary so much from book to book it was not possible to do more detailed tailoring. If this was required, the prepare-gb script might be a starting point.

The following example uses another edition of the Herodotus text, taking a single page from a Google Books edition as input:

```
cat in/gb_pg234.html | scripts/run -t gb -g plplus -top -o out gb_pg234
```

The output files are similar to those already shown and are available in the out directory.

The Open Library provides an alternative source of scanned and OCR-ed texts, and experiments were also done with material from this source. The text displays many of the same OCR errors but is available as plain text (as well as other formats less useful for processing) rather than html.

4.4.2 XML input

The “172172.txt” input used above was actually originally generated as xml, in the “ltgxml” format - the plain text version has the markup stripped out. In the ltgxml format the docdate can be included if known:
Nadal and Murray set up semi showdown

(CNN) -- Rafael Nadal and Andy Murray are both through to the semifinals of the Rogers Cup in Toronto, where they will face each other for a place in Sunday’s final.

Murray played some superb tennis in crushing the in-form David Nalbandian but Nadal had to recover from dropping the opening set to get past Germany’s Philipp Kohlschreiber.

Nalbandian won the ATP title in Washington last weekend and came into Friday’s encounter on an 11-match unbeaten streak.

...
CHAPTER
FIVE

THE PIPELINE

The geoparser is implemented in modular fashion, as a sequence of steps arranged in a “pipeline”. The aim is to make it easy to switch different components in if desired, for instance if a local POS tagger is preferred to the one supplied here.

As illustrated in Figure Overview of the geoparser pipeline, there are two stages to the geoparsing process:

1. Geotagging
2. Georesolution

![Figure 5.1: Overview of the geoparser pipeline](image)

The geotagging step process input text to identify and classify named entities within it, specifically placename entities though other classes can also be found - see The nertag Component.

The georesolution step uses a gazetteer (see Gazetteers) to ground placename entities against specific geographic locations mentioned in the gazetteer. Typically there will be multiple candidates - for example, there are any number of places called “Edinburgh” in the world. The georesolver ranks the candidates in order using various contextual clues.

5.1 Geotagging

NOTE: THIS CHAPTER HAS BEEN AUTOMATICALLY CONVERTED FROM LATEX AND NEEDS TIDYING - TO BE DONE ONCE REWRITES (IN LATEX) ARE COMPLETE.
5.1.1 Introduction

This documentation is intended to provide a detailed description of the pipelines provided in the LT-TTT2 distribution. The pipelines are implemented as Unix shell scripts and contain calls to processing steps which are applied to a document in sequence in order to add layers of XML mark-up to that document.

This document does not contain any explanation of grammars or XPath expressions. For an introduction to the grammar rule formalism, see the tutorial documentation (draft) in . See also [http://www.cogsci.ed.ac.uk/~richard/ltxml2/lxtransduce-manual.html](http://www.cogsci.ed.ac.uk/~richard/ltxml2/lxtransduce-manual.html) as well as the documentation for the LT-XML2 programs at [http://www.cogsci.ed.ac.uk/~richard/ltxml2/](http://www.cogsci.ed.ac.uk/~richard/ltxml2/).

LT-TTT2 includes some software not originating in Edinburgh which has been included with kind permission of the authors. Specifically, the part-of-speech (POS) tagger is the C&C tagger and the lemmatiser is . See Sections [postag] and [lemmatise] for more information and conditions of use.

LT-TTT2 also includes some resource files which have been derived from a variety sources including UMLS, Wikipedia, Project Gutenberg, Berkeley and the Alexandria Digital Library Gazetteer. See Sections [tokenise], [lemmatise] and [nertag] for more information and conditions of use.

5.1.2 Pipelines

The run script

The LT-TTT2 pipelines are found in the directory and are NLP components or sub-components, apart from which is a pipeline that applies all of the NLP components in sequence to a plain text document. The diagram in Figure shows the sequence of commands in the pipeline.

The script is used from the command line in the following kinds of ways (from the directory):

```
./scripts/run < data/example1.txt > your-output-file
```

```
cat data/example1.txt | ./scripts/run | more
```

The steps in Figure appear in the script as follows:

```
```

Step 1 copies the input to a temporary file, (see Section [setup] for information about ). This is then used in Step 2 as the input to the first processor which converts a plain text file to XML and writes its output as the temporary file. Each successive step takes as input the temporary file which is output from the previous step and writes its output to another appropriately named temporary file. The output of the final processor is written to and the final step of the pipeline uses the Unix command to send this file to standard output.
Setup

[setup]

All of the pipeline scripts contain this early step:

```
. 'dirname $0'/setup
```

This causes the commands in the file to be run at this point and establishes a consistent naming convention for paths to various resources. For the purposes of understanding the content of the pipeline scripts, the main points to note are:

- The variable takes as value the full path to the directory.
- A variable is defined as and is then added to the value of the user’s variable so that the scripts can call the executables such as without needing to specify a path.
- The variable is defined for use by the scripts to write temporary files and ensure that they are uniquely named. The value of follows this pattern: 
  `/tmp/<USERNAME>-<NAME-OF-SCRIPT>-<PROCESS-ID>`. Thus the temporary file created by Step 2 above (the temporary file containing the output of ) might be .

Temporary files are removed automatically after the script has run, so cannot usually be inspected. Sometimes it is useful to retain them for debugging purposes and the setup script provides a method to do this—if the environment variable is set then the temporary files are not removed. For example, this command:

```
LXDEBUG=1 ./scripts/run <data/example1.txt >testout.xml
```

causes the script to be run and retains the temporary files that are created along the way.

Component Scripts

The main components of the pipeline as shown in Figure The run pipeline are also located in the directory. They are described in detail in Sections [sec:preparetxt]–[sec:chunk].

The needs of users will vary and not all users will want to use all the components. The script has been designed so that it is simple to edit and configure for different needs. There are dependencies, however:

- assumes a plain text file as input;
- all other components assume an XML document as input;
- requires its input to contain paragraphs marked up as elements;
- the output of contains (sentence) and (word) elements and all subsequent components require this format as input;
- , and require part-of-speech (POS) tag information so must be applied before them;
- if both and are used then should be applied before .

Each of the scripts has the effect of adding more XML mark-up to the document. In all cases, except , the new mark-up appears on or around the character string that it relates to. Thus words are marked up by wrapping word strings with a element, POS tags and lemmas are realised as attributes on elements, and named entities are marked up by wrapping sequences with appropriate elements. The script allows the user to choose among a variety of output formats, including BIO column format and standoff output (see Section [sec:chunk] for details). Section [sec:visualise] discusses how the XML output of pipelines can be converted to formats which make it easier to visualise.

The components are Unix shell scripts where input is read from standard input and output is to standard output. Most of the scripts have no arguments apart from and : details of their command line options can be found in the relevant sections below.

The component scripts are similar in design and in the beginning parts they follow a common pattern:

- and variables are defined for use in error reporting;

5.1. Geotagging
5.1.3 The `preparetext` component

Overview

The component is a Unix shell script called with no arguments. Input is read from standard input and output is to standard output.

This script converts a plain text file into a basic XML format and is a necessary step since the LT-XML2 programs used in all the following components require XML as input. The script generates an XML header and wraps the text with a text element. It also identifies paragraphs and wraps them as elements. If the input file is this:

```
This is a piece of text.
It needs to be converted to XML.
```

the output is this:

```
<?xml version="1.0" encoding="ISO-646"?>
<!DOCTYPE text [
<!ELEMENT text (#PCDATA)+>
]>
<p>This is a piece of text.</p>
<p>It needs to be converted to XML.</p>
```

Some users may want to process data which is already in XML, in which case this step should not be used. Instead, it should be ensured that the XML input files contain paragraphs wrapped as elements. So long as there is some kind of paragraph mark-up, this can be done using `lxreplace`. For example, a file containing `para` elements like this:

```
<body><para>This is a piece of text.</para>
<para>It needs to be converted to XML.</para></body>
```

can easily be converted using this command:

```
cat input-file | lxreplace -q para -n "'p'"
```

so that the output is this:

```
<body><p>This is a piece of text.</p>
<p>It needs to be converted to XML.</p></body>
```

Note that parts of the XML structure above the paragraph level do not need to be changed since the components only affect either paragraphs or sentences and words inside paragraphs.

The `preparetext` script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute
The pipeline.

**The `preparetext` pipeline**

```
& lxplain2xml -e guess -w text | 2. & lxtransduce -q text $lib/paras.gr
```

**Step 1: “`lxplain2xml -e guess -w text`“**

This step uses the LT-XML2 program to convert the text into an XML file. The output is the text wrapped in a text root element (\texttt{-w text}) with an XML header that contains an encoding attribute which guesses (\texttt{-e guess}) based on the characters it encounters in the text. The output of this step given the previous input file is this:

```
<?xml version="1.0" encoding="ISO-646"?>
<!DOCTYPE text [ 
<!ELEMENT text (#PCDATA)*> ]> 
<text>
This is a piece of text.
</text>
```

The file contains a UTF-8 pound character. If Step 1 is used with this file as input, the output has a UTF-8 encoding:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE text [ 
<!ELEMENT text (#PCDATA)*> ]> 
<text>
This example contains a UTF-8 character, i.e. \"£\".
</text>
```

**Step 2: “`lxtransduce -q text $lib/paras.gr`“**

The second and final step in the pipeline uses the LT-XML2 program with the grammar rule file to identify and mark up paragraphs in the text as elements. On the first example in this section the output contains two paragraphs as already shown above. On a file with no paragraph breaks, the entire text is wrapped as a element, for example:

```
<?xml version="1.0" encoding="ISO-646"?>
<!DOCTYPE text [ 
<!ELEMENT text (#PCDATA)*> ]> 
<p>This is a piece of text. It needs to be converted to XML.</p>
</text>
```

Note that if the encoding is UTF-8 then the second step of the pipeline does not output the XML declaration since UTF-8 is the default encoding. Thus the output of on the file is this:

```
<!DOCTYPE text [ <!ELEMENT text (#PCDATA)*> ]> <text><p>This example contains a UTF-8 character, i.e. \"£\".</p></text>
```

**5.1.4 The `tokenise` Component**

[tokenise]

**Overview**

The component is a Unix shell script called with no arguments. Input is read from standard input and output is to standard output.

**5.1. Geotagging**
This is the first linguistic processing component in all the top level scripts and is a necessary prerequisite for all other linguistic processing. Its input is an XML document which must contain paragraphs marked up as elements. The component acts on the elements by (a) segmenting the character data content into (word) elements and (b) identifying sentences and wrapping them as elements. Thus an input like this:

```xml
<document>
  <text>
    <p>
      This is an example. There are two sentences.
    </p>
  </text>
</document>
```

is transformed by and output like this (modulo white space which has been changed for display purposes):

```xml
<document>
  <text>
    <p>
      <s id="s1"
        w id="w3" c="w" pws="yes">This</s>
      <s id="s2"
        w id="w23" c="w" pws="yes">There</s>
      <w id="w11" c="w" pws="yes">an</w>
      <w id="w14" c="w" pws="yes">example</w>
      <w id="w21" pws="no" sb="true">.</w>
    </p>
    <s id="s2"
      w id="w29" c="w" pws="yes">are</s>
    <w id="w33" c="w" pws="yes">two</w>
    <w id="w37" c="w" pws="yes">sentences</w>
    <w id="w46" pws="no" sb="true">.</w>
  </text>
</document>
```

The attribute on elements encodes a unique id for each word based on the start position of its first character. The attribute on elements encodes unique sequentially numbered ids for sentences. The attribute is used to encode word type (see Table [conv] for complete list of values). It serves internal purposes only and can possibly be removed at the end of preprocessing. All elements have a attribute which has a value if there is no white space between the word and the preceding word and a value otherwise. The attribute on sentence final full stops serves to differentiate these from sentence internal full stops. The and attributes are used by the component.

### The tokenise script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.

#### The tokenise pipeline

```
& lxtransduce -q p $lib/pretokenise.gr | 2. & lxtransduce -q p $lib/tokenise.gr | 3. & lxreplace -q "w/cg" | 4. & lxtransduce -q p -l lex=$lib/mobyfuncwords.lex $lib/sents-news.gr | 5. & lxtransduce -q s -l lex=$here/lib/nertag/numbers.lex $lib/posttokenise.gr | 6. & lxreplace -q "w/w" | 7. & lxreplace -q "w[preceding-sibling::*[1][self::w]]" -t "<w pws='no'>\attrs;\children;</w>" | 8. & lxreplace -q "w[not(@pws)]" -t "<w pws='yes'>\attrs;\children;</w>" | 9. & lxreplace -q cg | 10. & lxaddids -e 'w' -p "w" -c '/text()' | 11. & lxaddids -e 's' -p "'s'
```

**Step 1:** “lxtransduce -q p $lib/pretokenise.gr”
The first step in the pipeline uses with the rules in . The query (-q p) establishes elements as the part of the XML that the rules are to be applied to. The pretokenise grammar converts character data inside elements into a sequence of 'character groups' ( elements) so that this:

```xml
<p>"He’s gone", said Fred.</p>
```

is output as follows:

```xml
<p><cg c='qut' qut='d'>"</cg><cg c='lca'>H</cg><cg c='lca'>e</cg><cg c='lca'>s</cg><cg c='ws'> </cg><cg c='lca'>g</cg><cg c='ws'> </cg><cg c='nl'></cg></p>
```

Note that here and elsewhere we introduce line breaks to display examples to make them readable but that they are not to be thought of as part of the example. Every actual character in this example is contained in a , including whitespace and newline characters, e.g. the newline between said and Fred in the current example. The attribute on elements encodes the character type, e.g. indicates lower case. Table [concg] contains a complete list of values for the attribute on elements. Note that quote elements () have a further attribute to indicate whether the quote is single or double: or .

**[h]**

<table>
<thead>
<tr>
<th>ampersand</th>
<th>percent sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>bracket (round, square, brace)</td>
<td>quote</td>
</tr>
<tr>
<td>digits</td>
<td>forward and backward slashes</td>
</tr>
<tr>
<td>comma, colon, semi-colon</td>
<td>full stop, question mark, exclamation mark</td>
</tr>
<tr>
<td>single dash, sequence of dashes</td>
<td>symbols such as , @ etc.</td>
</tr>
<tr>
<td>sequence of dots</td>
<td>tab character</td>
</tr>
<tr>
<td>greater than (character or entity)</td>
<td>upper case alphabetic</td>
</tr>
<tr>
<td>lowercase alphabetic</td>
<td>lowercase n’t</td>
</tr>
<tr>
<td>lowercase n’t</td>
<td>unknown characters</td>
</tr>
<tr>
<td>less than entity</td>
<td>whitespace</td>
</tr>
</tbody>
</table>

[concg]

**Step 2: \"lxtransduce -q p $lib/tokenise.gr\"**

The second step in the pipeline uses with . The query again targets elements but in this step the grammar uses the elements of the previous step and builds elements from them. Thus the output of step 1 is converted to this:

```xml
<p><w c="lquote" qut="d"><cg qut="d">"</cg></w><w c="uca">H</w><w c="lca">e</w><w c="aposs">s</w><w c="ws"> </w><w c="w"><cg c="lca">g</cg></w><w c="ws"> </w><cg c="cm">, </cg><w c="w"><cg c="lca">F</cg></w><cg c="nl"></cg></p>
```

Note that the apostrophe+s sequence in He’s has been recognised as such ( value for the attribute). Non-apostrophe quote elements acquire an , or value for (left, right or can’t be determined) and have a further attribute to indicate whether the quote is single or double: or . Table [conw] contains a complete list of values for the attribute on elements.

[h]
Step 3: “lxreplace -q “w/cg””

The third step uses to remove elements inside the new elements. (Word internal elements are no longer needed, but those occurring between words marking whitespace and newline are retained for use by the sentence grammar.) The output now looks like this:

```xml
<p><w qut="d" c="lquote">"</w><w c="w">He</w><w c="aposs">’</w><cg c="ws"> </cg><w c="w">gone</w><w qut="d" c="rquote">"</w><w c="cm">,</w><cg c="ws"> </cg><w c="w">said</w><cg c="nl">.
</cg></p>
```

Step 4: “lxtransduce -q p -l lex=$lib/mobyfuncwords.lex $lib/sents-news.gr”

The next step uses to mark up sentences as elements. As well as using the rule file, a lexicon of function words (derived from Project Gutenberg’s Moby Part of Speech List) is consulted. This is used as a check on a word with an initial capital following a full stop: if it is a function word then the full stop is a sentence boundary. The output on the previous example is as follows:

```xml
<p><s><w c="lquote" qut="d">"</w><w c="w">He</w><w c="aposs">’</w><cg c="ws"> </cg><w c="w">gone</w><w qut="d" c="rquote">"</w><w c="cm">,</w><cg c="ws"> </cg><w c="w">said</w><cg c="nl">.
</cg></s></p>
```

The script is set up to use a sentence grammar which is quite general but which is tuned in favour of newspaper text and the abbreviations that occur in general/newspaper English. The distribution contains a second sentence grammar, which is essentially the same grammar but which has been tuned for biomedical text. For example, the abbreviation Mr. or MR. is expected not to be sentence final in but is permitted to occur finally in . Thus this example:

```xml
<p>
I like Mr. Bean.
XYZ interacts with 123 MR. Experiments confirm this.
</p>
```

is segmented by as:

```xml
<p>
<s>I like Mr.</s><s>Bean.</s>
<s>XYZ interacts with 123 Mr.</s>
<s>Experiments confirm this.</s>
</p>
```

while segments it like this:

```xml
<p>
<s>I like Mr.</s><s>.</s>
<s>Bean.</s><s>.</s>
<s>XYZ interacts with 123 MR.</s><s>.</s>
<s>Experiments confirm this.</s><s>.</s>
</p>
```

---

1 http://www.gutenberg.org/etext/3203
The grammar has been tested on the Genia corpus and performs very well.

**Step 5:** “lxtransduce -q s l lex=$here/lib/nertag/numbers.lex $lib/posttokenise.gr”

The fifth step applies with the rule file to handle hyphenated words and to handle full stops belonging to abbreviations. Since an layer of annotation has been introduced by the previous step, the query now targets elements rather than elements. In the input to , hyphens are split off from their surrounding words, so this grammar combines them to treat most hyphenated words as words rather than as word sequences—it wraps a element (with the attribute ) around the relevant sequence of elements, thus creating inside mark-up. The grammar consults a lexicon of numbers in order to exclude hyphenated numbers from this treatment. (Later processing by the numex and timex named entity rules requires that these should be left separated.) Thus if the following is input to :

```xml
<p>Mr. Bean eats twenty-three ice-creams.</p>
```

the output after the post-tokenisation step is:

```xml
<p><s><w c="abbr">Mr</w><w c="."/>.w><cg c="ws"> </cg><w c="w">Bean</w></s></p>
```

The grammar also handles full stops which are part of abbreviations by wrapping a element (with the attribute ) around a sequence of a word followed by a non-sentence final full stop (thus again creating elements). The Mr. in the current example demonstrates this aspect of the grammar.

Note that this post-tokenisation step represents tokenisation decisions that may not suit all users for all purposes. Some applications may require hyphenated words not to be joined (e.g. the biomedical domain where entity names are often subparts of hyphenated words (NF-E2-related)) and some downstream components may need trailing full stops not to be incorporated into abbreviations. This step can therefore be omitted altogether or modified according to need.

**Step 6:** “lxreplace -q “w/w”"

The sixth step in the pipeline uses to remove the embedded mark-up in the multi-word words created in the previous step.

**Step 7 & 8:**

```bash
lxreplace -q "w[preceding-sibling::*:1][self::w]" -t "w pws='no'&attrs;&children;/w" | lxreplace -q "w[not(@pws)]" -t "w pws='yes'&attrs;&children;/w"
```

The seventh and eighth steps add the attribute to elements. This attribute indicates whether the word is preceded by whitespace or not and is used by other, later LT-TTT2 components (e.g., the component). Step 7 uses to add to elements whose immediately preceding sibling is a . Step 8 then adds to all remaining elements.

**Step 9:** “lxreplace -q cg”

At this point the mark-up is no longer needed and is removed by step 9. The output from steps 6–9 is as follows:

```xml
<p>Mr. Bean eats twenty-three ice-creams.</p>
```

**Steps 10 & 11:**

In the final two steps is used to add id attributes to words and sentences. The initial example in this section, reproduced here, shows the input and output from where the words and sentences have acquired ids through these final steps:

```xml
5.1. Geotagging
In step 10, the part of the command prefixes the id value with \texttt{w}. The option ensures that the numerical part of the id reflects the position of the start character of the element (e.g. the initial \textit{e} in \textit{example} is the 14th character in the element). We use this kind of id so that retokenisations in one part of a file will not cause id changes in other parts of the file. Step 11 is similar except that for id values on elements the prefix is \texttt{s}. We have also chosen not to have the numerical part of the id reflect character position—instead, through not supplying a option, the default behaviour of sequential numbering obtains.

5.1.5 The \texttt{postag} Component

[postag]

Overview

The component is a Unix shell script called with one argument via the option. The argument to is the name of a model directory. The only POS tagging model provided in this distribution is the one found in but we have parameterised the model name in order to make it easier for users wishing to use their own models. Input is read from standard input and output is to standard output.

POS tagging is the next step after tokenisation in all the top level scripts since other later components make use of POS tag information. The input to is a document which has been processed by and which contains , , and elements. The component adds a attribute to each with a value which is the POS tag assigned to the word by the C&C POS tagger using the model. Thus an input like this (output from ):
The POS tagger called by the script is the C&C maximum entropy POS tagger trained on data tagged with the Penn Treebank POS tagset. We have included the relevant Linux binary and model from the C&C release at http://svn.ask.it.usyd.edu.au/trac/candc/wiki with the permission of the authors. The binary of the C&C POS tagger, which in this distribution is named , is a copy of from the tar file. The model, which in this distribution is named , is a copy of from the tar file. This model was trained on the Penn Treebank (see for more details). The C&C POS tagger may be used under the terms of the academic (non-commercial) licence at http://svn.ask.it.usyd.edu.au/trac/candc/wiki/Licence.

Note that the script is simply a wrapper for a particular non-XML based tagger. It converts the input XML to the input format of the tagger, invokes the tagger, and then merges the tagger output back into the XML representation. It is possible to make changes to the script and the conversion files in order to replace the C&C tagger with another.

The postag script

Since is called with an argument, the early part of the script is more complex than scripts with no arguments. The and loops set up the argument so that the path to the model has to be provided when the component is called. Thus all the top level scripts which call the component do so in this way:

$here/scripts/postag -m $here/models/pos

In the next part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.

The postag pipeline

```
1. & cat >$tmp-in2 & lxconvert -w -q s -s $lib/pos.cnv <$tmp-in | 3. & pos -model $model 2>$tmp-ccposerr | 4. & lxconvert -r -q s -s $lib/pos.cnv -x $tmp-in

Step 1: “cat >$tmp-in”
```

The first step in the pipeline copies the input to the temporary file. This is so that it can both be converted to C&C input format as well as retained as the file that the C&C output will be merged with.

5.1. Geotagging
Step 2: ```lxconvert -w -q s -s $lib/pos.cnv <$tmp-in```  

The second step uses to convert into the right format for input to the C&C POS tagger (one sentence per line, tokens separated by white space). The option instructs it to use the stylesheet, while the query makes it focus on elements. (The component will therefore not work on files which do not contain elements.) The option makes it work in write mode so that it follows the rules for writing C&C input format. If the following output:

```xml
<p><s id="s1"><w id="w0" c="abbr" pws="yes">Mr.</w> <w id="w4" c="w" pws="yes">Bean</w> <w id="w8" c="w" pws="yes">had</w> <w id="w12" c="w" pws="yes">an</w> <w id="w15" c="hyw" pws="yes">ice-cream</w><w id="w25" pws="no" sb="true" c="."/>.w</s><s id="s2"><w id="w27" c="w" pws="yes">He</w> <w id="w30" c="w" pws="yes">dropped</w> <w id="w34" c="w" pws="yes">it</w><w id="w40" pws="no" sb="true" c="."/>.w</s></p>
```

is input to the first step, its output looks like this:

Mr. Bean had an ice-cream .
He dropped it .

and this is the format that the C&C POS tagger requires.

Step 3: ```pos -model $model 2>$tmp-ccposerr```  

The third step is the one that actually runs the C&C POS tagger. The command has a option and the argument to that option is provided by the variable which is set by the option of the script, as described above. The 2>$tmp-ccposerr ensures that all C&C messages are written to a temporary file rather than to the terminal. If the input to this step is the output of the previous step shown above, the output of the tagger is this:

Mr.|NNP Bean|NNP had|VBD an|DT ice-cream|NN .|.
He|PRP dropped|VBD it|PRP .|.

Here each token is paired with its POS tag following the ‘|’ separator. The POS tag information in this output now needs to be merged back in with the original document.

Step 4: ```lxconvert -r -q s -s $lib/pos.cnv -x $tmp-in```  

The fourth and final step in the component uses with the same stylesheet as before () to pair the C&C output file with the original input which was copied to the temporary file, , in step 1. The option to identifies this original file. The option tells to use read mode so that it follows the rules for reading C&C output (so as to cause the POS tags to be added as the value of the attribute on elements). The query again identifies elements as the target of the rules. For the example above which was output from the previous step, the output of this step is as follows:

```xml
<p><w pws="yes" c="abbr" id="w0" p="NNP">Mr.</w> <w pws="yes" c="w" id="w4" p="VBD">Bean</w> <w pws="yes" c="w" id="w8" p="NN">had</w> <w pws="yes" c="w" id="w12" p="DT">an</w> <w pws="yes" c="hyw" id="w15" p="NN">ice-cream</w> <w c="." sb="true" pws="no" id="w25" p="."/>.w</s><s id="s2"><w pws="yes" c="w" id="w27" p="PRP">He</w> <w pws="yes" c="w" id="w30" p="VBD">dropped</w> <w pws="yes" c="w" id="w34" p="PRP">it</w> <w c="." sb="true" pws="no" id="w40" p="."/>.w</s></p>
```

5.1.6 The **lemmatise** Component

[lemmatise]  

Overview  

The component is a Unix shell script called with no arguments. Input is read from standard input and output is to standard output.
The component computes information about the stem of inflected words: for example, the stem of *peas* is *pea* and the stem of *had* is *have*. In addition, the verbal stem of nouns and adjectives which derive from verbs is computed: for example, the verbal stem of *arguments* is *argue*. The lemma of a noun, verb or adjective is encoded as the value of the attribute on elements. The verbal stem of a noun or adjective is encoded as the value of the attribute on elements.

The input to is a document which has been processed by and and which therefore contains , , and elements with POS tags encoded in the attribute of elements. Since lemmatisation is only applied to nouns, verbs and verb forms which have been tagged as adjectives, the syntactic category of the word is significant—thus the component must be applied after the component and not before. When the following is passed through and :

```xml
<document>
  <text>
    <p>The planning committee were always having big arguments.
The children have frozen the frozen peas.</p>
  </text>
</document>
```

it is output like this (again modulo white space):

```xml
<document>
  <text>
    <p>
      <s id="s1">The</s> planning committee were always having big arguments.<n id="w3" p="DT">The</n>
      <w id="w7" p="NN">committee</w> were always having big arguments.<n id="w16" p="NN">committee</n> were always having big arguments.<n id="w26" p="VBD">were</n>
      <w id="w31" p="RB">were</w> were always having big arguments.<n id="w38" p="VBD">were</n>
      <w id="w45" p="RB">were</w> were always having big arguments.<n id="w49" p="VBD">were</n>
      <w id="w58" p="RB">were</w> were always having big arguments.<n id="w60" p="DT">The</n>
      <w id="w64" p="NN">children</w> have frozen the frozen peas.<n id="w73" p="DT">The</n>
      <w id="w78" p="NNS">children</w> have frozen the frozen peas.<n id="w85" p="DT">have</n>
      <w id="w96" p="VBN">frozen</w> the frozen peas.<n id="w100" p="DT">have</n>
      <w id="w114" p="VBN">frozen</w> the frozen peas.<n id="w118" p="DT">have</n>
      <w id="w131" p="VBN">frozen</w> the frozen peas.<n id="w135" p="DT">have</n>
      <w id="w148" p="VBN">frozen</w> the frozen peas.<n id="w152" p="DT">have</n>
      <w id="w166" p="VBN">frozen</w> the frozen peas.<n id="w170" p="DT">have</n>
      <w id="w184" p="VBN">frozen</w> the frozen peas.<n id="w188" p="DT">have</n>
      <w id="w192" p="VBN">frozen</w> the frozen peas.<n id="w196" p="DT">have</n>
      <w id="w200" p="VBN">frozen</w> the frozen peas.<n id="w204" p="DT">have</n>
    </p>
  </text>
</document>
```

The lemmatiser called by the script is . We have included the relevant binary and verb stem list from the release at [http://www.informatics.susx.ac.uk/research/groups/nlp/carroll/morph.html](http://www.informatics.susx.ac.uk/research/groups/nlp/carroll/morph.html) with the permission of the authors. The binary of , which in this distribution is located at , is a copy of from the tar file . The resource file, , which in this distribution is located in the directory is copied from the same tar file. The software is free for research purposes.

Note that the script is similar to the script in that it is a wrapper for a particular non-XML based program. It converts the input XML to the input format of the lemmatiser, invokes the lemmatiser, and then merges its output back into the XML representation. It is possible to make changes to the script and the conversion files in order to plug out the lemmatiser and replace it with another. The pipeline does a little more than just wrap however, because it also computes the attribute on certain nouns and adjectives (see step 4 in the next section). In doing this it uses a lexicon of information about the verbal stem of nominalisations (e.g. the stem of *argument* is *argue*). This lexicon, , is derived from the file in the 2007 UMLS SPECIALIST lexicon distribution .

The lemmatise script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.

The lemmatise pipeline

1. & cat >$tmp-in 2. & lxconvert -w -q w -s $lib/lemmatise.cnv <$tmp-in 3. & morpha -f $lib/verbstem.list 4. & lxconvert -r -q w -s $lib/lemmatise.cnv -x $tmp-in

Step 1: cat >$tmp-in

The first step in the pipeline copies the input to the temporary file. This is so that it can both be converted to input format as well as retained as the file that the output will be merged with.

Step 2: lxconvert -w -q w -s $lib/lemmatise.cnv <$tmp-in

The second step uses to convert into an appropriate format for input to the lemmatiser (one or sometimes two word_postag pairs per line). The option instructs it to use the stylesheet, while the query makes it focus on elements. (The component will therefore work on any file where words are encoded as elements and POS tags are encoded in the attribute on .) The option makes it work in write mode so that it follows the rules for writing input format. If the following output:

```
<p>
<s id="s1">
  <w pws="yes" c="w" id="w3" p="DT">The</w> <w pws="yes" c="w" id="w7" p="NN">planning</w> <w pws="yes" c="w" id="w16" p="NN">committee</w> <w pws="yes" c="w" id="w26" p="VBD">were</w> <w pws="yes" c="w" id="w31" p="RB">always</w> <w pws="yes" c="w" id="w42" p="VBG">having</w> <w c="." sb="true" pws="no" id="w58" p="."/>
</s>
<s id="s2">
  <w pws="yes" c="w" id="w60" p="DT">The</w> <w pws="yes" c="w" id="w64" p="NNS">children</w> <w pws="yes" c="w" id="w73" p="VBP">have</w> <w pws="yes" c="w" id="w78" p="VBN">frozen</w> <w pws="yes" c="w" id="w85" p="DT">the</w> <w pws="yes" c="w" id="w89" p="JJ">frozen</w> <w pws="yes" c="w" id="w96" p="NNS">peas</w> <w c="." sb="true" pws="no" id="w100" p="."/>
</s>
</p>
```

is input to the first step, its output looks like this:

```
planning_NN planning_V
committee_NN were_VBD
having_VBG
big_JJ
arguments_NNS
carriage_NNS
have_VBP
frozen_VBN
frozen_JJ frozen_V
peas_NNS
```

Chapter 5. The Pipeline
Each noun, verb or adjective is placed on a line and its POS tag is appended after an underscore. Where a noun or an adjective ends with a verbal inflectional ending, a verb instance of the same word is created (i.e.,) in order that its output for the verb can be used as the value for the attribute.

**Step 3: morpha -f $lib/verbstem.list**

The third step is the one that actually runs. The command has a option to provide a path to the resource file that it uses. If the input to this step is the output of the previous step shown above, the output of is this:

```
planning plan
committee
be
have
big
argument
child
have
freeze
frozen freeze
pea
```

Here it can be seen how the POS tag affects the performance of the lemmatiser. The lemma of planning is planning when it is a noun but plan when it is a verb. Similarly, the lemma of frozen is frozen when it is an adjective but freeze when it is a verb. Irregular forms are correctly handled (children:child, frozen:freeze).

**Step 4: lxconvert -r -q w -s $lib/lemmatise.cnv -x $tmp-in**

The fourth and final step in the component uses with the same stylesheet as before () to pair the output file with the original input which was copied to the temporary file, , in step 1. The option to identifies this original file. The option tells to use read mode so that it follows the rules for reading output. The query again identifies elements as the target of the rules. For the example above which was output from the previous step, the output of this step is as follows (irrelevant attributes suppressed):

```
<p><s><w p="DT">The</w> <w p="NN" l="planning" vstem="plan">planning</w></s></p>
<s><w p="NN" l="committee">committee</w><w p="VBD" l="be">were</w></s>
<s><w p="RB">always</w> <w p="VBG" l="have">having</w><w p="JJ">big</w></s>
<s><w p="NNS" l="argument" vstem="argue">arguments</w><w p=".">.</w></s></p>
<s><w p="DT">The</w> <w p="NNS" l="child">children</w><w p="have">have</w><w p="VBN" l="freeze">frozen</w></s>
<s><w p="DT">the</w> <w p="JJ">frozen</w> <w p="NNS" l="pea">peas</w><w p=".">.</w></s></p>
```

Here the lemma is encoded as the value of and, where a second verbal form was input to (planning, frozen as an adjective), the output becomes the value of the attribute. Whenever the lemma of a noun can be successfully looked up in the nominalisation lexicon (), the verbal stem is encoded as the value of (argument:argue). The relevant entry from is this:

```
<lex word="argument" stem="argue"/>
```

### 5.1.7 The nertag Component

[nertag]

**Overview**

[nerintro]
The component is a Unix shell script called with no arguments. Input is read from standard input and output is to standard output.

The component is a rule-based named entity recogniser which recognises and marks up certain kinds of named entity: numex (sums of money and percentages), timex (dates and times) and enamex (persons, organisations and locations). These are the same entities as those used for the MUC7 named entity evaluation. (In addition also marks up some miscellaneous entities such as urls.)

Unlike the other components, has a more complex structure where it makes calls to subcomponent pipelines which are also located in the directory. Figure The nertag pipeline shows the structure of the nertag pipeline.

![Diagram of the nertag pipeline]

Figure 5.3: The nertag pipeline

The input to is a document which has been processed by , and and which therefore contains , , and elements and the attributes , and on the elements. The rules identify sequences of words which are entities and wrap them with the elements , and , with subtypes encoded as the value of the attribute. For example, the following might be input to a sequence of , and .

```xml
<document>
  <text>
    <p>
      Peter Johnson, speaking in London yesterday afternoon, said that profits for ABC plc were up 5% to $17 million.
    </p>
  </text>
</document>
```

The output is a relatively unreadable XML document where all the , , and elements and attributes described in the previous sections have been augmented with further attributes and where , and elements have been added. For clarity we show the output below after and mark up has been removed using the command . Removing extraneous mark-up in this way and at this point might be appropriate if named entity recognition was the final aim of the processing. If further processing such as chunking is to be done then the and mark-up must be retained.

```xml
<document>
  <text>
    <s id="s1">Peter Johnson</s>, speaking in London yesterday afternoon, said that profits for ABC plc were up 5% to $17 million.</text>
</document>
```
The nertag script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains a sequence of processing steps piped together:


(is defined in the setup as the directory). Unlike previous components, these steps are calls to subcomponents which are themselves shell scripts containing pipelines. Thus the process is sub-divided into three subcomponents, to identify and mark up and elements, to apply dictionary lookup for names and, finally, which marks up elements taking into account the output of the following subsections describe each of these subcomponents in turn.

Note that the grammars used in the subcomponent are updated versions of the grammars used in and previously distributed in the original LT-TTT distribution. The output of is therefore of relatively high quality. The other two subcomponents are new for this release and the rules have not been extensively tested or tuned.

The numtimex script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.

The numtimex pipeline

1. & lxtransduce -q s -l lex=$lib/numbers.lex $lib/numbers.gr | 2. & lxreplace -q "phr/phr" | 3. & lxreplace -q "phr[w][count(node())=1]" -t "children;" | 4. & lxtransduce -q s -l lex=$lib/currency.lex $lib/numex.gr | 5. & lxreplace -q "phr[not(@c='cd') and not(@c='yrrange') and not(@c='frac')]" | 6. & lxtransduce -q s -l lex=$lib/timex.lex -l numlex=$lib/numbers.lex $lib/timex.gr | 7. & lxreplace -q "phr[not(.~' ')]" -t & "<w><xsl:apply-templates select='w[1]/@*'/>&attrs;<xsl:value-of select='.'/></w>"

Step 1: lxtransduce -q s -l lex=$lib/numbers.lex $lib/numbers.gr

Numerical expressions are frequent subparts of and entities so the first step in the pipeline identifies and marks up a variety of numerical expressions so that they are available for later stages of processing. This step uses with the rules in the grammar file and uses the query so as to process the input sentence by sentence. It consults a lexicon of number words () which contains word entries for numbers (e.g. eighty, billion). If the following sentence is processed by step 1 after first having been put through and (and but this doesn’t affect and is disregarded here):

The third announcement said that the twenty-seven billion euro deficit was discovered two and a half months ago.

the output will be this (again modulo white space):

<!-- Begin Highlights -->

```
<p><s id="s1"><w p="DT" id="w1" c="w" pws="yes">The</w><br />
<phr c="ord"><w p="JJ" id="w5" c="ord" pws="yes">three</w></phr><br />
<w p="NN" id="w11" c="w" pws="yes">announcement</w><br />
<w p="VBD" id="w24" c="w" pws="yes">said</w><br />
<w p="IN" id="w38" c="cd" pws="yes">twenty</w><br />
<w p="CD" id="w44" c="cd" pws="no" c="hyph">)</w>
```

5.1. Geotagging
This can be seen more clearly if we remove the elements:

Step 2: lxreplace -q "phr/phr"

The second step uses to remove embedded mark-up so that numerical phrases don’t have unnecessary internal structure:

Step 3: lxreplace -q "phr[w][count(node())=1]" -t "&children;"

The third step makes another minor adjustment to the mark-up. The grammar will sometimes wrap single words as elements (e.g. the third in the current example) and since this is unnecessary, in this step is used to remove any tag where there is a single daughter. Thus the current example is changed to this:

Step 4: lxtransduce -q s -l lex=$lib/currency.lex $lib/numex.gr

The fourth step of the pipeline recognises entities using the rules in . It is this step which is responsible for the two instances of mark-up in the example in section [nerintro]. For the current example, the output of this step (after removing elements) is this:

The grammar makes use of the lexicon which contains a list of the names of a wide range of currencies. Using this information it is able to recognise the money element.

Step 5: lxreplace -q "phr[not(@c='cd') and not(@c='yrange') and not(@c='frac')]"

It is not intended that mark-up should be part of the final output of a pipeline—it is only temporary mark-up which helps later stages and it should be deleted as soon as it is no longer needed. At this point, elements with , and as values for the attribute are still needed but other elements are not. This step removes all elements which are not still needed.
The sixth step of the pipeline recognises entities using the rules in . It is this step which is responsible for the two instances of mark-up in the example in section [nerintro]. For the current example, the output of this step (after removing elements) is this:

```xml
<p><s id="s1">The third announcement said that the <numex type="money"><phr c="cd">twenty-seven billion</phr> euro</numex> deficit was discovered <timex type="date"><phr c="cd">two and a half</phr> months ago</timex>.</s></p>
```

The grammar makes use of two lexicons, , which contains entries for the names of days, months, holidays, time zones etc., and . In addition to examples of the kind shown here, the timex rules recognise standard dates in numerical or more verbose form (08/31/07, 31.08.07, 31st August 2007 etc.), times (half past three, 15:30 GMT etc.) and other time related expressions (late Tuesday night, Christmas, etc.).

Step 7:

```bash
lxreplace -q "phr[not(\sim' ')]]" -t "<w><xsl:apply-templates select='w[1]/@*'/>&attrs;<xsl:value-of select='.'/></w>"
```

By this point the only mark-up that will still be needed is that around multi-word phrases, i.e. those containing white space (e.g. three quarters). Where there is no white-space, this step creates a element instead of the original . The new element acquires first the attributes of the first in the old () and then the attributes of the old itself ()—since both have a attribute, the one from the is retained. The text content of the embedded elements are copied but the embedded element tags are not. The following is an example of input to this step. Note that the line break between three and is there for layout purposes and does not exist in the actual input.

```xml
<p><s id="s1"><phr c="cd"><w pws="yes" c="cd" id="w1" p="CD">two</w><w pws="yes" c="cd" id="w5" p="CD">thousand</w></phr><w c="cm" pws="no" id="w13" p=":">;</w><phr c="frac"><w pws="yes" c="cd" id="w15" p="CD">three</w><w c="hyph" pws="no" id="w20" p="-"/>-quarters</phr></s></p>
```

The output for this example is this:

```xml
<p><s id="s1"><phr c="cd"><w p="CD" id="w1" c="cd" pws="yes">two</w><w p="CD" id="w5" c="cd" pws="yes">thousand</w></phr><w p=":" id="w13" pws="no" c="cm">;</w><w p="CD" id="w15" c="frac" pws="yes">three-quarters</w></s>
```

The result is that three-quarters is now recognised as a single word token, rather than the three from before. This brings the mark-up more into line with standard tokenisation practise which does not normally split hyphenated numbers: subsequent steps can therefore assume standard tokenisation for such examples. The two thousand example is left unchanged because standard tokenisation treats this as two tokens. However, since we have computed that together two and thousand constitute a numerical phrase, we keep the mark-up for future components to benefit from. For example a noun group chunking rule can describe a numeric noun specifier as either a or a instead of needing to make provision for one or more numeric words in specifier position. If, however, the component is to be the last in a pipeline and no further LT-TTT2 components are to be used, either the last step can be changed to remove all mark-up or the call to can be followed by a call to to remove elements.

The lexlookup script

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.
The lexlookup pipeline

1. & lxtransduce -q s -a firstname $lib/lexlookup.gr

Step 1: lxtransduce -q s -a firstname $lib/lexlookup.gr

This step uses to mark up words which are known forenames. The option to instructs it to apply the rule:

   <rule name="firstname" attrs="pername='true'">
     <first>
       <lookup match="w[@p~'^N' and .~'^[A-Z]'" lexicon="fname" phrase="true"/>
       <lookup match="w[@p~'^N' and .~'^[A-Z]'" lexicon="mname" phrase="true"/>
     </first>
   </rule>

This rule does look-up against two lexicons of female and male first names where the locations of the lexicons are defined in the grammar like this:

<lexicon name="fname" href="femalefirstnames.lex"/>
<lexicon name="mname" href="malefirstnames.lex"/>

i.e. the lexicons are expected to be located in the same directory as the grammar itself. The lexicons are derived from lists at http://www.ssa.gov/OACT/babynames/.

This step adds the attribute to words which match so that

<w p="NNP">Peter</w>

becomes

<w p="NNP" pername="true">Peter</w>.

2. & lxtransduce -q s -a common $lib/lexlookup.gr

Step 2: lxtransduce -q s -a common $lib/lexlookup.gr

This step uses to identify capitalised nominals which are known to be common words. The option to instructs it to apply the rule:

   <rule name="common" attrs="common='true'">
     <lookup match="w[@p~'^N' and .~'^[A-Z]'" lexicon="common" phrase="true"/>
   </rule>

This rule does look-up against a lexicon of common words where the location of the lexicon is defined in the grammar like this:

<lexicon name="common" href="common.mmlex"/>

i.e. the lexicon is expected to be located in the same directory as the grammar itself. The common word lexicon is derived from an intersection of lower case alphabetic entries in Moby Part of Speech (http://www.gutenberg.org/etext/3203) and a list of frequent common words derived from available from the Berkeley Web Term Document Frequency and Rank site (http://elib.cs.berkeley.edu/docfreq/). Because this is a very large lexicon (25,307 entries) it is more efficient to use a memory-mapped version (with an extension) since the default mechanism for human-readable lexicons loads the entire lexicon into memory and incurs a significant start-up cost if the lexicon is large. Memory-mapped lexicons are derived from standard lexicons using the LT-XML2 program. The source of , , is located in the directory and can be searched. If it is changed, the memory-mapped version needs to be recreated.

The effect of step 2 is to add the attribute to capitalised nominals which match so that
Step 3: `lxtransduce -q s -a otherloc $lib/lexlookup.gr`

This step uses to identify the names of countries (e.g. France) as well as capitalised words which are adjectives or nouns relating to place names (e.g. French). The option to instructs it to apply the rule:

```xml
<rule name="otherloc">
  <first>
    <lookup match="w[.~'^[A-Z]'\]">
      lexicon="countries" phrase="true" attrs="country='true'"/>
    <lookup match="w[\@[p~'^[NJ]' and .~'^[A-Z]'\]">
      lexicon="locadj" phrase="true" attrs="locadj='true'"/>
  </first>
</rule>
```

The first lookup in the rule accesses the lexicon of country names while the second accesses the lexicon of locational adjectives, where the location of the lexicons are defined in the grammar like this:

```xml
<lexicon name="locadj" href="locadj.lex"/>
<lexicon name="countries" href="countries.lex"/>
```

i.e. the lexicons are expected to be located in the same directory as the grammar itself. The lexicons are derived from lists at http://en.wikipedia.org/wiki/United_Nations_member_states and http://en.wikipedia.org/wiki/List_of_adjectival_forms_of_place_names.

The effect of step 3 is to add the attributes and to capitalised words which match so that

```xml
<w p="NN">Portuguese</w> and <w p="NN">Brazil</w>
```

become

```xml
<w p="NN" locadj="true">Portuguese</w> and <w p="NN" country="true">Brazil</w>.
```

Step 4: `lxtransduce -q s -a place $lib/lexlookup.gr`

The final step uses to identify the names of places. The option to instructs it to apply the rule:

```xml
<rule name="place">
  <first>
    <ref name="place-multi"/>
    <ref name="place-single"/>
  </first>
</rule>
```

This accesses two rules, one for multi-word place names and one for single word place names. For multi-word place names, the assumption is that these are unlikely to be incorrect, so the rule wraps them as:

```xml
<rule name="place-multi" wrap="enameX" attrs="type='location'">
  <and>
    <query match="w[.~'^[A-Z]'\]">
      <first>
        <lookup match="w" lexicon="alexm" phrase="true"/>
        <lookup match="w[\@[p~'^N' and .~'^[A-Z]+$']">
          lexicon="alexm" case="no" phrase="true"/>
        </first>
      </and>
    </first>
  </and>
</rule>
```
Single word place names are highly likely to be ambiguous so the rule for these just adds the attribute to words which match.

```
<rule name="place-single" attrs="locname='single'">
  <and>
    <query match="w[^'^[A-Z]']"/>
    <first>
      <lookup match="w" lexicon="alexs" phrase="true"/>
      <lookup match="w[@p~'^N' and .~'^[A-Z][A-Z][A-Z][A-Z]+$']" lexicon="alexs" case="no" phrase="true"/>
    </first>
  </and>
</rule>
```

These rules access lexicons of multi-word and single-word place names, where the location of the lexicons are defined in the grammar like this:

```
<lexicon name="alexm" href="alexandria-multi.mmlex"/>
<lexicon name="alexs" href="alexandria-single.mmlex"/>
```

i.e. the lexicons are expected to be located in the same directory as the grammar itself. The source of the lexicons is the Alexandria Digital Library Project Gazetteer (http://www.alexandria.ucsb.edu/gazetteer/), specifically, the name list, which can be downloaded from http://www.alexandria.ucsb.edu/downloads/gazdata/adlgaz-namelist-20020315.tar

Various filters have been applied to the list to derive the two separate lexicons, to filter common words out of the single-word lexicon and to discard certain kinds of entries. As with the common word lexicon, we use memory-mapped versions of the two lexicons because they are very large (1,797,719 entries in and 1,634,337 entries in).

The effect of step 4 is to add mark-up or to words which match so that

```
<w p="NNP">Manhattan</w>
```

becomes

```
<w p="NNP" locname="single">Manhattan</w>
```

and

```
<w p="NNP">New</w> <w p="NNP">York</w>
```

becomes

```
<enamex type="location"><w pername="true">New</w> <w p="NNP">York</w></enamex>.
```

Note that because the rules in are applied in a sequence of calls rather than all at once, a word may be affected by more than one of the look-ups. See, for example, the words Robin, Milton and France in the output for Robin York went to the British Rail office in Milton Keynes to arrange a trip to France.:

```
<s><w common="true" pername="true">Robin</w> <w locname="single">York</w> <w went</w> <w to</w> <w the</w> <w locadj="true">British</w> <w Rail</w> <w office</w> <w in</w> <w><enamex type="location"><w pername="true">Milton</w> <w Keynes</w></enamex> <w to</w> <w arrange</w> <w a</w> <w trip</w> <w to</w> <w locname="single" country="true">France</w> </s>.
```

The new attributes on elements are used by the rules in the component, while the multi-word location mark-up prevents these entities from being considered by subsequent rules. Thus Milton Keynes will not be analysed as a person name.

**The enamex script**

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute

---

3 This list is available for download and local use within the limits of the ADL copyright statement, which is reproduced in the LT-TTT2 distribution as.
the pipeline.

**The enamex pipeline**

1. & lxtransduce -q s -l lex="$lib/enamex.lex" $lib/enamex.gr | 2. & lxreplace -q "enamex/enamex" > $tmp-pre-otf 3. & $here/scripts/onthefly <$tmp-pre-otf >$tmp-otf.lex 4. & lxtransduce -q s -l lex=$tmp-otf.lex $lib/enamex2.gr <$tmp-pre-otf | 5. & lxreplace -q subname

**Step 1: lxtransduce -q s -l lex="$lib/enamex.lex" $lib/enamex.gr**

Step 1 in the pipeline applies the main grammar, which marks up elements of type , and , as well as miscellaneous entities such as urls. An input like this:

```
<p>
Mr. Joe L. Bedford (www.jbedford.org) is President of JB Industries Inc.
Bedford has an office in Paris, France.
</p>
```

is output as this (mark-up suppressed):

```
<p>
<s id="s1"><enamex type="person">Mr. Joe L. Bedford</enamex> (<url>www.jbedford.org</url>) is President of <enamex type="organization">JB Industries Inc</enamex>.</s>
<s id="s2">Bedford has an office in Paris, <enamex type="location">France</enamex>.</s>
</p>
```

At this stage, single-word place names are not marked up as they can be very ambiguous—in this example Bedford is a person name, not a place name. The country name France, has been marked up, however, because the component marked it as a country and country identification is more reliable.

**Step 2: lxreplace -q "enamex/enamex" > $tmp-pre-otf**

Multi-word locations are identified during and can form part of larger entities, with the result that it is possible for step 1 to result in embedded marked, e.g.:

```
<enamex type="organization"><enamex type="location">Bishops Stortford</enamex> Town Council</enamex>
```

Since embedded mark-up is not consistently identified, it is removed. This step applies to remove inner mark-up. The output of this step is written to the temporary file because it feeds into the creation of an ‘on the fly’ lexicon which is created from the first pass of in order to do a second pass matching repeat examples of first pass entities.

**Step 3: $here/scripts/onthefly <$tmp-pre-otf >$tmp-otf.lex**

The temporary file from the last step, is input to the script (described in Sections [otfscript] and [otfpipe]) which creates a small lexicon containing the elements which have already been found plus certain variants of them. If the example illustrating step 1 is input to , the lexicon which is output is as follows:

```
<lexicon>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="France"><cat>location</cat></lex>
<lex word="JB Industries Inc"><cat>organization</cat></lex>
<lex word="Joe"><cat>person</cat></lex>
<lex word="Joe Bedford"><cat>person</cat></lex>
```

5.1. Geotagging
Step 4: `lxtransduce -q s -l lex=tmp-otf.lex $lib/enamex2.gr <$tmp-pre-otf`

The ‘on the fly’ lexicon created at step 3 is used in step 4 with a second enamex grammar. This performs lexical lookup against the lexicon and in our current example this leads to the recognition of *Bedford* in the second sentence as a person rather than a place. The grammar contains a few other rules including one which finally accepts single word placenames () as locations—this results in *Paris* in the current example being marked up.

Step 5: `lxreplace -q subname`

The final step of the component (and of the component) is one which removes a level of mark-up that was created by the rules in the grammar, namely the element. This was needed to control how a person name should be split when creating the ‘on the fly’ lexicon, but it is no longer needed at this stage. The final output of the component for the current example is this:

```xml
<p><s id="s1"><enamex type="person">Mr. Joe L. Bedford</enamex> (<url>www.jbedford.org</url>) is President of <enamex type="organization">JB Industries Inc</enamex>.
<s id="s2"><enamex type="person" subtype="otf">Bedford</enamex> has an office in <enamex type="location">Paris</enamex>, <enamex type="location">France</enamex>.</s></p>
```

**The onthefly script**

`[otfscript]`

This script uses the LT-XML2 programs to extract names from the first pass of and convert them into an ‘on the fly’ lexicon (the lexicon referred to above). The conversion is achieved through sequences of and as well as use of and. This is a useful example of how simple steps using these programs can be combined together to create a more complex program.

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute the pipeline.

**The onthefly pipeline**

`[otfpipe]`

```
ll
1. & lxgrep -w lexicon & enamex[@type='person' and not(subname[@type='fullname'])] & |subname[@type='fullname']|enamex[@type='location']|enamex[@type='organization']" | 2. & lxreplace -q "enamex" -t "<name>&attrs;&children;</name>" | 3. & lxreplace -q "w[@*]" | 4. & lxreplace -q "name/subname" -t "<w>&children;</w>" | 5. & lxreplace -q "w/" | 6. & lxreplace -q "lexicon/subname" -t "<name type='person'>&children;</name>" | 7. & lxreplace -q "lexicon/*/text()" -r "normalize-space(.)" | 8. & lxreplace -q "w{~'^(.|[A-Z]\d$)'}" -t "<w init='yes'>&children;</w>" | 9. & lxreplace -q "w[.~'^(.|[A-Z]\d$)']" -t "<w init='yes'>&children;</w>" | 10. & lxreplace -q "w[position()!=1]" -t "<xsl:text> </xsl:text>" | 11. & lxreplace -q w | 12. & lxreplace -q name | 13. & lxreplace -q name -t "<lex word='(.).*<cat>\xsl:value-of select='@type'/></cat></lex>" | 14. & lxsort lex icon lex | 15. & lxsort lex icon lex @word | 16. & lxuniq lex icon lex @word | 17. & lxuniq lex cat | 18. & lxuniq lex cat.
```
Step 1

The first step uses to extract location and organization elements as well as either full person elements or a relevant subpart of a name which contains a title. The input is a document with , , , and , and mark-up and the output of this call to for the previous Mr. Joe L. Bedford example is this:

```
<lexicon>
  <subname type="fullname">
    <w pername="true" l="joe" id="w4" c="w" pws="yes" p="NNP" locname="single">Joe</w>
    <w l="bedford" id="w8" c="w" pws="yes" p="NNP" locname="single">Bedford</w>
  </subname>
  <enamex type="organization">
    <w l="jb" id="w51" c="w" pws="yes" p="NNP">JB</w>
    <w l="industry" id="w54" c="w" pws="yes" p="NNPS" common="true">Industries</w>
    <w l="inc" id="w65" c="w" pws="yes" p="NNP">Inc</w>
  </enamex>
  <enamex type="location">
    <w country="true" l="france" id="w102" c="w" pws="yes" p="NNP" locname="single">France</w>
  </enamex>
</lexicon>
```

Steps 2–8

The next seven steps use to gradually transform the and elements in the output into elements: The elements inside the elements lose their attributes and the white space between them is removed (because the original white space in the source text may be irregular and include newlines). In Step 8, elements which are initials are given the attribute so that they can be excluded from consideration when variants of the entries are created. The output from these five steps is this:

```
<lexicon>
  <name type="person"><w>Joe</w><w init="yes">L.</w><w>Bedford</w></name>
  <name type="organization"><w>JB</w><w>Industries</w><w>Inc</w></name>
  <name type="location"><w>France</w></name>
</lexicon>
```

Step 9

Step 9 uses with the stylesheet to create extra variant entries for person names. The output now looks like this:

```
<lexicon>
  <name type="person"><w>Joe</w><w init="yes">L.</w><w>Bedford</w></name>
  <name type="person"><w>Bedford</w></name>
  <name type="person"><w>Joe</w></name>
  <name type="person"><w>Joe</w></name>
  <name type="person"><w>Bedford</w></name>
  <name type="person"><w>Bedford</w></name>
  <name type="person"><w>Joe</w><w>Bedford</w></name>
  <name type="organization"><w>JB</w><w>Industries</w><w>Inc</w></name>
  <name type="location"><w>France</w></name>
</lexicon>
```

The duplicates are a side-effect of the rules in the stylesheet and are removed before the end of the pipeline.

Steps 10–13

The next four steps use to continue the transformation of the elements. Regular white space is inserted between the elements and then the mark up is removed. Any empty elements are removed and the conversion to proper lexicon format is done with the final . The output now looks like this:

```
5.1. Geotagging
```
<lexicon>
<lex word="Joe L. Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Joe"><cat>person</cat></lex>
<lex word="Joe"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="Joe Bedford"><cat>person</cat></lex>
<lex word="Joe Bedford"><cat>person</cat></lex>
<lex word="JB Industries Inc"><cat>organization</cat></lex>
<lex word="France"><cat>location</cat></lex>
</lexicon>

**Step 14**

At this stage there are still duplicates so this step uses with the stylesheet
to add to each entry the elements of all its duplicates. The output
from this step looks like this:

<lexicon>
<lex word="Joe L. Bedford"><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat><cat>person</cat><cat>person</cat></lex>
<lex word="Joe"><cat>person</cat><cat>person</cat></lex>
<lex word="Joe"><cat>person</cat><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat><cat>person</cat><cat>person</cat></lex>
<lex word="Bedford"><cat>person</cat><cat>person</cat><cat>person</cat></lex>
<lex word="JB Industries Inc"><cat>organization</cat></lex>
<lex word="France"><cat>location</cat></lex>
</lexicon>

Note that in this example, each entity is only of one type. In other examples, the same string may have been
identified by the enamex grammar as belonging to different types in different contexts, for example, *Prof. Ireland
happens to work in Ireland*. In this case the output at this stage looks like this:

<lexicon>
<lex word="Ireland"><cat>person</cat><cat>location</cat></lex>
<lex word="Ireland"><cat>person</cat><cat>location</cat></lex>
</lexicon>

**Steps 15–18**

The final four steps of the pipeline use and to remove duplicate entries and duplicate elements. The final result for
the running example is this:

<lexicon>
<lex word="Bedford"><cat>person</cat></lex>
<lex word="France"><cat>location</cat></lex>
<lex word="JB Industries Inc"><cat>organization</cat></lex>
<lex word="Joe"><cat>person</cat></lex>
<lex word="Joe Bedford"><cat>person</cat></lex>
<lex word="Joe L. Bedford"><cat>person</cat></lex>
</lexicon>

**5.1.8 The chunk Component**

[sec:chunk]
Overview

[chunkintro]

The component is a Unix shell script. Input is read from standard input and output is to standard output. The script requires two parameters supplied through and options. The option specifies the style of output that is required with possible arguments being: , , or . The option specifies the format of output with possible arguments being: , or .

The component is a rule-based chunker which recognises and marks up shallow syntactic groups such as noun groups, verb groups etc. A description of an earlier version of the chunker can be found at . The earlier version only marked up noun and verb groups while the current version also marks up preposition, adjective, adverb and sbar groups. The first part of the pipeline produces mark-up which is similar to, though not identical to, the chunk mark-up in the CoNLL 2000 data. This mark-up is then converted to reflect different chunking styles and different formats of output through use of the and parameters.

The output of the first part of the pipeline, when applied after tokenisation and POS tagging, converts this input:

In my opinion, this example hasn’t turned out well.

to this output (whitespace altered):

<text>
<p><s id="s1">
<pg><w p="IN" pws="yes" id="w1">In</w></pg>
</s><ng>
<w p="PRP$" pws="yes" id="w4">my</w>
<w p="NN" pws="yes" id="w7" headn="yes">opinion</w>
</ng>
<w p="," pws="no" id="w14">,</w>
</pg>
<w p="DT" pws="yes" id="w16">this</w>
<w p="NN" pws="yes" id="w21" headn="yes">example</w>
</ng>
</pg>
<vg tense="pres" voice="act" asp="perf" modal="no" neg="yes">has</vg>
<w p="RB" pws="no" id="w32" neg="yes">n’t</w>
<vg tense="pres" voice="act" asp="perf" modal="no" neg="yes">turned</vg>
<rg><w p="RB" pws="yes" id="w47">well</w></rg>
<w p="." sb="true" pws="no" id="w51">.</w></s></p>
</text>

Note that elements have attributes indicating values for tense, aspect, voice, modality and negation and that head verbs and nouns are marked as and respectively. These attributes are extra features which are not normally output by a chunker but which are included in this one because it is relatively simple to augment the rules for these features.

The effects of the different style and format options are described below.

The chunk rules require POS tagged input but can be applied before or after lemmatisation. The component would typically be applied after the component since the rules have been designed to utilise the output of ; however, the rules do not require output and the chunker can be used directly after POS tagging.

The chunk script

Since is called with arguments, the early part of the script is more complex than scripts with no arguments. The and loops) set up the and options so that style and format parameters can be provided when the component is called. For example, the run script calls the component in this way:

$here/scripts/chunk -s nested -f inline

In the early part of the script the variable is defined to point to which is the location of the resource files used by the pipeline. The remainder of the script contains the sequence of processing steps piped together that constitute

5.1. Geotagging
the basic pipeline as well as conditional processing steps which format the output depending on the the choice of values supplied to the and parameters.

**The chunk pipeline**

1. & lxtransduce -q s $lib/verbg.gr |
2. & lxreplace -q "vg[w[@neg='yes']]
  -t "<vg neg='yes'>&attrs;&children;</vg>
3. & lxtransduce -q s $lib/noung.gr |
4. & lxtransduce -q s -l lex=$lib/other.lex $lib/otherg.gr |
5. & lxreplace -q "phr[@c" > $tmp-chunked

**Step 1: lxtransduce -q s $lib/verbg.gr**

The first step applies a grammar to recognise verb groups. The verb groups are wrapped as elements and various values for attributes encoding tense, aspect, voice, modality, negation and the head verb are computed. For example, the verb group from the previous example is output from this step as follows:

\[
<vg modal="no" asp="perf" voice="act" tense="pres">
  <w id="w29" c="w" pws="yes" p="VBZ" headv="yes">has</w>
  <w neg="yes" id="w32" neg="yes" id="w32" c="w" p="RB">n't</w>
  <w headv="yes" id="w36" c="w" pws="yes" p="VBN">turned</w>
  <w id="w43" c="w" pws="yes" p="RP">out</w>
</vg>
\]

The element contains the attributes , , and while the attribute occurs on the head verb and a attribute occurs on any negative words in the verb group.

**Step 2: lxreplace -q "vg[w[@neg='yes']]
  -t "<vg neg='yes'>&attrs;&children;</vg>"**

In the second step, information about negation is propagated from a negative word inside a verb group to the enclosing element. Thus the previous example now looks like this:

\[
<vg tense="pres" voice="act" asp="perf" modal="no" neg="yes">
  <w p="VBZ" c="w" pws="yes" c="w" id="w29">has</w>
  <w p="RB" c="w" pws="no" id="w32" neg="yes">n't</w>
  <w p="VBN" c="w" pws="yes" c="w" id="w36">turned</w>
  <w p="RP" c="w" pws="yes" c="w" id="w43">out</w>
</vg>
\]

**Step 3: lxtransduce -q s $lib/noung.gr**

In this step the noun group grammar is applied. Noun groups are wrapped as elements and the head noun is marked with the attribute —see for example the two noun groups in the current example in Section [chunkintro]. In the case of compounds, all the nouns in the compound are marked with the attribute:

\[
<ng>
  <w id="w1" c="w" pws="yes" p="DT">A</w>
  <w headn="yes" id="w3" c="w" pws="yes" p="NN">snow</w>
  <w headn="yes" id="w8" c="w" pws="yes" p="NN">storm</w>
</ng>
\]

In the case of coordination, the grammar treats conjuncts as separate noun groups if possible:
but where a noun group seems to contain a coordinated head then there is one noun group and all head nouns as well as conjunctions are marked as:

\[
\text{green eggs and blue ham}
\]

In this particular case, there is a genuine ambiguity as to the scope of the adjective \textit{green} depending on whether it is just the eggs that are green or both the eggs and the ham that are green. The output of the grammar does not represent ambiguity and a single analysis will be output which will sometimes be right and sometimes wrong. The output above gives \textit{green} scope over both nouns and therefore gives the second reading. This is appropriate for this case but would probably be considered wrong for \textit{red wine and cheese}.

The noun group grammar rules allow for the possibility that the text has first been processed by the component by defining \textit{, and} elements as possible sub-parts of noun groups. This means that the output of the noun group grammar may differ depending on whether has been applied or not. For example, the component identifies \textit{the Office for National Statistics} as an element and this is then treated by the noun group grammar as an:

\[
\text{the Office for National Statistics}
\]

When isn’t first applied, the chunker outputs the example as a sequence of noun group, preposition group, noun group:

\[
\text{the Office for National Statistics}
\]

\textbf{Step 4: \texttt{lxtransduce -q s -l lex=$lib/other.lex $lib/otherg.gr}}

The fourth step uses the grammar to identify all other types of phrases. The lexicon it consults is a small list of multi-word prepositions such as \textit{in addition to}. The grammar identifies preposition groups (), adjective groups (), adverb groups () and sbar groups () so the output for \textit{And obviously, over time, it seems that things get better.} is this ( mark up suppressed):

\[
5.1. \text{Geotagging}
\]
And obviously, over time, it seems that things get better.

The only words which are not part of a chunk are punctuation marks and occasional function words such as the And in this example. The heads of the chunks identified by are not marked as such though it would be fairly simple to do so if necessary.

Step 5: lxreplace -q "phr|@c" > $tmp-chunked

The fifth step is the final part of the chunking part of the pipeline. This step uses to discard mark-up which is no longer needed: elements were added by the component and are used by the chunk rules but can be removed at this point. The attribute on words is also no longer needed. The output at this stage is written to a temporary file, which is used as the input to the next steps in the pipeline which format the chunk output depending on the choices made with the and parameters.

Final steps: style and format

Through the parameter, the user can require the chunker output to conform to a particular style. The possible options for this parameter are, or. As described in, different people may make different assumptions about how to mark up more complex chunks and there is a difference between our assumptions and those behind the mark-up of the CoNLL chunk data. To make it easier to compare with CoNLL-style chunkers, the grammars in the previous steps of the pipeline create an initial chunk mark-up which can be mapped to the CoNLL style or to some other style. The option for causes this initial mark-up to be output. If the example Edinburgh University’s chunker output can be made to vary is first processed with the component so that Edinburgh University is marked up as an and is then processed by the following two steps:

$here/scripts/chunk -s none -f inline |
lxreplace -q w

then the output is as follows:

<s>
  <ng>
    <enamex type="organization">Edinburgh University</enamex>
  </ng>
  's chunker output</cng>
</ng>

The example contains a possessive noun phrase and a verb with an infinitival complement, which cause the main points of difference in style. The and elements have been created as temporary mark-up which can be modified in different ways to create different styles. CoNLL style is created through the following steps:

lxreplace -q cvg -t "<vg>&children;</vg>" |
lxreplace -q "vg/vg" |
lxreplace -q "ng[cng]" -t "&children;" |
lxreplace -q "cng" -t "<ng>&children;</ng>" |
lxreplace -q "ng[ng]" -t "&children;" |
lxreplace -q "numex|timex|enamex"
Here the embedded and the are output as elements while the embedded elements are discarded and the is mapped to a . Mark up created by (, and elements) is also discarded:

```xml
<s>
  <ng>Edinburgh University</ng>
  <ng>'s chunker output</ng>
  <vg>can be made to vary</vg>
</s>
```

An alternative non-hierarchical style is created using the option which causes the following steps to be taken:

```
lxreplace -q cvg |
lxreplace -q "cng/ng/ng" |
lxreplace -q "numex|timex|enamex"
```

Here the is removed and the embedded elements are retained while embedded mark up in elements is removed and mark-up is also removed:

```xml
<s>
  <ng>Edinburgh University's chunker output</ng>
  <vg tense="pres" voice="pass" asp="simple" modal="yes">can be made</vg>
  <vg tense="inf" voice="act" asp="simple" modal="no">to vary</vg>
</s>
```

The style is provided for users who prefer to retain a hierarchical structure and is achieved through the following steps:

```
lxreplace -q "cng" |
lxreplace -q "cvg" -n "'vg'"
```

The output of this style is as follows:

```xml
<s>
  <ng>
    <enamex type="organization">Edinburgh University</enamex>
    's chunker output
  </ng>
  <vg>
    <vg modal="yes" asp="simple" voice="pass" tense="pres">can be made</vg>
    <vg modal="no" asp="simple" voice="act" tense="inf">to vary</vg>
  </vg>
</s>
```

So far all the examples have used the option, however, two other options are provided, and . The option converts chunk element mark-up to attribute mark-up on elements using the ConLL BIO convention where the first word in a chunk is marked as beginning that chunk (e.g. for the first word of a noun group), other words in a chunk are marked as in that chunk (e.g. for non-initial words in a noun group) and words outside a chunk are marked as . These labels appear as values of the attribute on elements and the chunk element mark-up is removed. This conversion is done using the stylesheet . If the previous example is put through , the output is this (irrelevant attributes suppressed):

```xml
<w group="B-NP">Edinburgh</w>
<w group="I-NP">University</w>
<w group="I-NP">’s</w>
<w group="I-NP" headn="yes">chunker</w>
<w group="I-NP" output</w>
<w group="B-VP">can</w>
<w group="I-VP">be</w>
<w group="I-VP" made</w>
<w group="B-VP">to</w>
<w group="I-VP" vary</w>
```
Chunk-related attributes on words are retained (e.g. and ) but attributes on elements have been lost and would need to be mapped to attributes on head verbs if it was felt necessary to keep them. Note that BIO format is incompatible with hierarchical styles and an attempt to use it with the or styles will cause an error. If the format option is chosen the output can then be passed on for further formatting, for example to create non-XML output. The stylesheet has been included as an example and will produce the following column format:

Edinburgh NNP B-NP
University NNP I-NP
′s POS I-NP
chunker NN I-NP
output NN I-NP
can MD B-VP
be VB I-VP
made VBN I-VP
to TO B-VP
vary VB I-VP
.
.

The standoff format is included to demonstrate how NLP component mark-up can be encoded as standoff mark-up. If the previous example is put through , the output is this:

Using with the stylesheet , the chunk mark up is removed from its inline position and a new element is created as the last element inside the element. This contains , etc. elements. The text content of the elements in is a copy of the string that they wrapped when they were inline. The relationship between the elements in the text and the chunk elements in is maintained through the use of the and attributes whose values are the values of the start and end words of the chunk. If the style option is chosen then all levels of and mark-up are put in the element:

<standoff>
<ng sw="w1" ew="w32">Edinburgh University′s chunker output</ng>
<vg sw="w39" ew="w46" modal="yes" asp="simple" voice="pass" tense="pres">
  can be made
</vg>
<vg sw="w51" ew="w54" modal="no" asp="simple" voice="act" tense="inf">
  to vary
</vg>
</standoff>
can be made
</vg>
<vg sw="w51" ew="w54" tense="inf" voice="act" asp="simple" modal="no">
to vary
</vg>
</standoff>

5.1.9 Visualising output

[sec:visualise]

XML documents with many layers of annotation are often hard to read. In this section we describe ways in which the mark-up from the pipelines can be viewed more easily. Often, simple command line instructions can be useful. For example, the output of run can be piped through a sequence of LT-XML2 programs to allow the mark-up you are interested in to be more visible:

echo 'Mr. Joe L. Bedford (www.jbedford.org) is President of JB Industries Inc. Bedford opened an office in Paris, France in September 2007.' | ./run | lxreplace -q w | lxgrep "s/*"

This command processes the input with the script and then removes the word mark-up and pulls out the chunks (immediate daughters of ) so that they each appear on a line:

<ng><enamex type="person">Mr. Joe L. Bedford</enamex></ng>
<ng><url>www.jbedford.org</url></ng>
<vg tense="pres" voice="act" asp="simple" modal="no">is</vg>
<ng>President</ng>
<pg>of</pg>
<ng><enamex type="organization">JB Industries Inc</enamex></ng>
<ng><enamex type="person" subtype="otf">Bedford</enamex></ng>
<vg tense="past" voice="act" asp="simple" modal="no">opened</vg>
<ng>an office</ng>
<pg>in</pg>
<ng><enamex type="location">Paris</enamex></ng>
<ng><enamex type="location">France</enamex></ng>
<pg>in</pg>
<ng><timex type="date">September 2007</timex></ng>

Another approach to visualising output is to convert it to html for viewing in a browser. In we provide three style sheets, one to display mark-up (), one to display mark-up () and one to display both (). The following command:

echo 'Mr. Joe L. Bedford (www.jbedford.org) is President of JB Industries Inc. Bedford opened an office in Paris, France in September 2007.' | ./run | lxt -s ../lib/visualise/htmlnerandchunk.xsl > visualise.html

creates an html file, which when viewed in a browser looks like this:

5.2 Georesolution

The georesolution step takes the tagged text file as input and processes the location entities to give them spatial co-ordinates. The chosen gazetteer is queried to produce a list of candidate locations for each toponym and these are ranked, with the highest ranking one chosen to be shown as a green marker on the map display, or as the only marker if the -top option is used.

The tagged text file produced by the geotagging step contains further markup - for other entity categories besides location (person, organisation, time expressions) and for temporal events, which are expressed as binary relations between pairs of entities. Although obviously the geoparser’s main business is with spatial entities, the temporal
relations are processed at the end of the georesolution step, to produce a timeline display of events detected in the text.

The input file for this step is in a temporary file, labelled “tmp-temprel” in the flowcharts of the Overview chapter; see Georesolution flowchart. The actual file will be in the /tmp directory, with a name that includes the username of the process in which the script was run and a unique string generated from the name of the script that’s running and its process number, suffixed in this case with “temprel” to identify the content, e.g. “$USER-run-5648-temprel”. These temporary files are removed when the pipeline exits unless the $LXDEBUG environment variable is set, in which case they are kept for examination.

The final output file - written to $outdir.out.xml if -o outdir is specified and to stdout otherwise - is described at output file in the Practical Examples chapter, and there is an example file here (html documentation only). The “tmp-temprel” file differs only in respect of the location entities. In the unprocessed temprel file these look like this:

```xml
<ent type="location" id="rb6">
  <parts>
    <part sw="w148" ew="w148">Toronto</part>
  </parts>
</ent>
```

The georesolution step adds extra attributes to this element, from the Geonames gazetteer in this example:

```xml
<ent id="rb6" type="location" lat="43.7001138" long="-79.4163042"
in-country="CA" gazref="geonames:6167865" feat-type="ppl"pop-size="4612191">
  <parts>
    <part ew="w148" sw="w148">Toronto</part>
  </parts>
</ent>
```

This is the top-ranked candidate, http://www.geonames.org/6167865/toronto.html. The other candidates are listed in $outdir/gaz.xml - see example file here (html documentation only). In this example there were 20 candidates for Toronto, which is the maximum number the geoparser considers. The first five are shown below:

```xml
<placenames>
  <placename id="rb6" name="Toronto">
    <place rank="1" score="1.762934636" scaled_type="0.8" scaled_pop="0.9327814568" scaled_contained_by="0" scaled_contains="0" scaled_near="0"
in-cc="CA" long="-79.4163" lat="43.7001138" type="ppla" gazref="geonames:6167865" pop="4612191" clusteriness="870.3494166"clusteriness_rank="9" locality="0"distance-to-known="99999"/>
  </placename>
</placenames>
```
There is one <placename> element for each distinct placename found in the input document - note, not for each individual mention. If a place is mentioned multiple times in a document the geoparser assumes the same place is being talked about each time. Clearly there are examples where this would be an erroneous assumption, e.g. in the text snippet:

“Are we talking about London, England or London, Ontario?”

There is in fact a special rule to catch containment expressed in this co-ordinated way, but nevertheless the current version of the geoparser will only pick a single location for London (the first one, in England).

The rest of the output files produced if -o is specified are for visualisation in a browser.

The rest of this chapter looks at each step of the georesolution process in a little more detail: firstly the collection of candidate places from the gazetteer, then the ranking process and finally the production of display files.

5.2.1 Gazetteer Lookup

The run script calls another, named geoground, which carries out two tasks by calling further scripts. The first is gazetteer lookup, done by the geogaz script which calls a version of gazlookup tailored for the gazetteer and including the gazetteer name. So for example, if -g geonames were specified to the run script then gazlookup-geonames would be used at this point, whereas if Pleiades+ were required then gazlookup-plplus would be invoked.

If you look in the scripts directory you will find a collection of these gazlookup scripts, most being completely separate routines, needed because the connection methods and queries to be used differ greatly between different gazetteers. The “Unlock” option is an exception as it has three variants - “Unlock”, “OS” and “Natural Earth” (see -t and -g parameters) - but these can be dealt with by parameterisation within a single script, gazlookup-unlock. There are soft links to this script to cover the other two variants because, in order to make it straightforward to add new gazetteer options, the geogaz script looks for a script named gazlookup-$gaz, where “$gaz” is the -g $gaz command line parameter.

This means that to add a new gazetteer to the pipeline, all you need do is create a script named
**Edinburgh Geoparser Documentation, Release 1.0**

**gazlookup-newgaz** that handles the connection and querying appropriately, and returns a set of candidates formatted as required for the next stage; and then alter the run script to accept “newgaz” as a valid -g option. Of course, if the domain covered by the new gazetteer is completely new, then alterations to the geotagging stage would also be needed - as for example was the case when the Pleiades gazetteer of ancient places was added to cater for classical texts.

The input to the gazlookup-$gaz step is a list of the locations found in the input, extracted by an XSL stylesheet named extractlocs.xsl. The list is formatted as shown in this example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<placenames>
  <placename id="rb6" name="Toronto"/>
  <placename id="rb11" name="Germany"/>
  <placename id="rb14" name="Washington"/>
  <placename id="rb22" name="Montreal"/>
  <placename id="rb28" name="Wimbledon"/>
  <placename id="rb32" name="France"/>
</placenames>
```

The output of the gazetteer lookup is a collection of up to 20 candidate <place> nodes for each <placename>. The final step of the geogaz script is to sort and deduplicate - as explained above, the assumption is made that multiple references to the same toponym string within a single document are referring to the same place.

The output of this stage is in a temporary file suffixed “gazunres.xml”, following the naming conventions described above. An example is here (html documentation only). It contains feature information extracted from the gazetteer for each candidate location, to be used by the ranking algorithm. The first few lines for our example are as follows:

```xml
<placenames>
  <placename name="Toronto" id="rb6">
    <place name="Toronto" gazref="geonames:149454" type="ppl"
          lat="-4.9000000" long="38.1000000" in-cc="TZ" pop="0"/>
    <place name="Toronto" gazref="geonames:2146222" type="ppl"
          lat="-33.0000000" long="151.6000000" in-cc="AU" pop="0"/>
    <place name="Toronto" gazref="geonames:3535110" type="ppl"
          lat="22.7833300" long="-82.5000000" in-cc="CU" pop="0"/>
    <place name="Toronto" gazref="geonames:3666869" type="ppl"
          lat="8.4039600" long="-75.2790700" in-cc="CO" pop="0"/>
  ...
</placenames>
```

This example makes clear the need for ranking over a reasonable number of candidates, at least for a gazetteer like Geonames with so many candidates for most placenames. For Toronto, the first four places returned were in Tanzania, Austria, Cuba and Columbia. We are up to numbers 13 and 14 before Canadian places appear in the list. For many places Geonames will return an extremely long list; the geoparser truncates the results at 20, which will almost always include the right one and makes the ranking process manageable in terms of processing time.

### 5.2.2 Ranking

The ranking of the <place> candidates is done by the georesolve script. If the gazetteer supplies feature information the ranking makes use of it, for example preferring populated places (Geonames code “PPL”) over natural features, and preferring larger to smaller places (based on population size).

Apart from the attributes of the candidate places, the ranking algorithm considers their locations compared pairwise with each of the other places in the document. It will prefer places that cluster with other locations in the same document. For example, if most of the places mentioned in a text seem to be in Canada, a mention of “London” will probably be placed in Ontario rather than England.

If you know the geographical area that your input document deals with, you can specify either a locality circle or box using the -l or -lb command line options. These are explained in in the Quick Start chapter, Limiting geographical area: -l/lb. This is another factor that will be considered by the ranker, making it prefer locations in the area specified but still allowing the selection of places elsewhere that may be mentioned in the text. The “score” parameter can be used for weighting the degree of preference; if using this option it is probably best to...
experiment with different weights. The output of the georesolve ranking step is the $outdir/gaz.xml that was described above. It is a ranked list of <place> candidates for each <placename>. The candidates have the features from the gazetteer and the extra attributes added by the ranking algorithm, such as “clusteriness” referring to how well the places mention form a spatial group. The raw scores are scaled and combined to produce an overall “score” attribute, which in turn determines the “rank” for each candidate <place>. See the sample output here (html documentation only).

It is worth noting here that for various reasons including the clustering factor, the geoparser works better with short texts than very long ones. It was originally designed to handle large numbers of short text documents (roughly one page at a time) processed in a loop. If an attempt is made to process an entire book in one go, the ranking algorithm may be overloaded - pairwise comparisons of locations throughout the document may break it - and in any case the assumption about locality will probably be invalid. We advise that long texts are split into small parts, preferably into coherent chunks of narrative.

5.2.3 Formatting Output

If the -o outdir option is not specified then the output of the pipeline is written to standard out (and can of course be redirected to a file), and consists of a single xml <document> as described at output file in the Practical Examples chapter, with an example file here (html documentation only). The output is a tagged version of the input file, in standoff xml format, with the <document> node having <text> and <standoff> children (plus a metadata node).

The placenames are tagged entities within the text, appearing as <ent> nodes in the standoff section with pointers back to their position in the tokenised text. Only the top candidate for each place is included in this output, as a tagged entity, such as:

```
<ent id="rb6" type="location" lat="43.70011" long="-79.4163"
gazref="geonames:6167865" in-country="CA" feat-type="ppla"
pop-size="4612191">
  <parts>
    <part ew="w150" sw="w150">Toronto</part>
  </parts>
</ent>
```

The ranking detail is removed and only the most important gazetteer features are retained: the latitude and longitude co-ordinates, and (for Geonames which supplies them) the country and feature type codes and population.

If the -o outdir option is specified then the georesolution component has several extra steps, which are simply reformatting of all the output generated so far, using XSL stylesheets to produce a collection of files for visualising the output. These steps are illustrated on the Georesolution flowchart.

The “plainvis.xsl” stylesheet is used to format the input text as an html page with the toponyms highlighted; DEEP has a special version which adds links back to the source gazetteer. The gazmap script pulls this html page together with the xml list of candidate placename locations in the $outdir/gaz.xml file described earlier and adds a map display created by plotting the locations using Google Maps. The three components are combined in a single file named $outdir/display.html. Various examples are shown in the Practical Examples chapter, including Geoparser display file for news text input, which has the maps panel at the top (green markers for top candidates, red for others), the tagged text on the left and the $outdir/gaz.xml list on the right.

If the -top option is specified then an additional set of files is created, with only the top candidate locations (green markers) retained. Herodotus display file shows an example.

Finally, the timeline script takes the tagged file and produces a display highlighting all the entities found: names, organisations and time expressions as well as locations. It also extracts the events detected and, where these can be given a specific date, uses javascript to create a timeline visualisation using a Simile widget. Timeline file shows an example of the $outdir.timeline.html file. The events found are listed in $outdir.events.xml, which is in the format required by the Timeline widget, as illustrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
<data date-time-format="iso8601">
  <event start="2010-08-15T00:00:00Z" title="will face each other for a place in Sunday">
    Nadal and Murray set up semi showdown (CNN) -- Rafael Nadal and Andy
  </event>
</data>
```
Murray are both through to the semifinals of the Rogers Cup in Toronto, where they will face each other for a place in Sunday’s final.

The complete file for this example is here (html documentation only).

In summary, with the `-o out` option, the following files are created:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$out.out.xml</td>
<td>Main output: tagged and geogrounded text</td>
</tr>
<tr>
<td>$out.gaz.xml</td>
<td>Locations list</td>
</tr>
<tr>
<td>$out.gazlist.html</td>
<td>Locations list in html format</td>
</tr>
<tr>
<td>$out.gazmap.html</td>
<td>Locations plotted on Google maps</td>
</tr>
<tr>
<td>$out.geotagged.html</td>
<td>Geotagged text as html file</td>
</tr>
<tr>
<td>$out.display.html</td>
<td>3-panel display: map + text + locations list</td>
</tr>
<tr>
<td>$out.gazlist-top.html</td>
<td>Top-ranked candidate list in html format</td>
</tr>
<tr>
<td>$out.gazmap-top.html</td>
<td>Top-ranked locations plotted on Google maps</td>
</tr>
<tr>
<td>$out.display-top.html</td>
<td>3-panel display: map + text + top-locations list</td>
</tr>
<tr>
<td>$out.nertagged.xml</td>
<td>Output from NER stage</td>
</tr>
<tr>
<td>$out.events.xml</td>
<td>Events extracted in Timeline format</td>
</tr>
<tr>
<td>$out.timeline.html</td>
<td>Display page with all NEs and timeline</td>
</tr>
</tbody>
</table>

The three “*-top*” files are only produced if the `-top` option is used.
6.1 Online Resources

The geoparser allows the user to choose from several different online gazetteers as the source authority against which to ground placenames. All except Geonames are hosted by Edina through the Unlock services; see The Edina Unlock Service chapter. In fact Unlock Places also maintains a mirror of Geonames but the geoparser is configured to go directly to the http://www.geonames.org site.

When the pipeline is executed using the run command (see Running the Pipeline) the gazetteer to be used must be specified using the -g parameter. The complete set of six online gazetteer options is as follows:

- Geonames, -g geonames - a world-wide gazetteer of over eight million placenames, made available free of charge.
- OS, -g os - a detailed gazetteer of UK places, derived from the Ordnance Survey 1:50,000 scale gazetteer, under the OS Open Data initiative. The geoparser code adds
- Natural Earth, -g naturalearth - a public domain vector and raster map collection of small scale (1:10m, 1:50m, 1:110m) mapping, built by the Natural Earth project.
- Unlock, -g unlock - a comprehensive gazetteer mainly for the UK, using both OS and Natural Earth resources and augmented with major worldwide cities and countries. This is the default option on the Unlock Places service and combines all their gazetteers except DEEP.
- Pleiades+, -g plplus - a gazetteer of the ancient Greek and Roman world, based on the Pleiades dataset and augmented with links to Geonames.

It may be necessary to experiment with different gazetteer options to see what works best with your text.

Pleiades+

The Pleiades gazetteer of the classical Greek and Roman world was added to the geoparser’s resources as part of the GAP project in 2012-13. The version used was a snapshot of the Pleiades source dataset augmented with links to Geonames - this was dubbed Pleiades+. This static copy of the data is mirrored on Edina and available with the -g plplus option. A locally hosted copy of it at Edinburgh’s School of Informatics was used by GAP.

Too late for the GAP project, the Pleiades dataset has been considerably augmented and daily snapshots are now available - see the Pleiades data download page. The Pleiades+ project - to align ancient places with their modern equivalents in Geonames where possible - has also been extended, and also provides daily downloads from the Pleiades Plus Github site. The organising teams behind both of these developments have kindly agreed that other sites can mirror their datasets, and Edina and we (the Language Technology Group) are hoping to do that. If you are interested in using this data and would like to help us update the geoparser service for it, please get in touch.

We have experimented with setting up a local copy of the latest version of Pleiades and Pleiades+ privately on the LTG servers, and the scripts to allow the -g plplus-local option, which accesses this local copy, are included in the distribution as explained below.
6.2 Options for Local Gazetteers

The standard way to use the geoparser is by referring to an online gazetteer service, as described above. There may be circumstances in which a locally hosted gazetteer is preferable - for example if the online service is slow, for the multiple hits required by the pipeline. The Edinburgh Language Technology Group (LTG) have set up local gazetteers in this way and this section explains how to do it. If you decide to do it, these models may be helpful to follow.

The advantages of hosting your gazetteer yourself are that access will typically be much faster so overall processing times are reduced, and you have complete control over the gazetteer so can correct errors or add new items. It may be necessary to have a local copy if your usage rates are so high that you exceed the limits placed by online services. The obvious disadvantage is that you create a maintenance burden for yourself, as you need to create and manage a database and write the software routines to interact with it.

The two local gazetteers we use are a local copy of Geonames and of Pleiades+. The setup of these is described below, as examples of how to go about the process. The code for these local gazetteers is included in the geoparser download bundle but it is not possible to access the local MySQL databases on our servers remotely, as they are not configured as public services.

6.2.1 Example Setup: Geonames

The Geonames service includes a download option with daily updates provided on their download server. The Geonames database is large - around 8 million main entries plus alternative name records - and the online service provides update files so that insertions and deletions can be applied to a local copy, without having to recreate and re-index the tables every day.

In the LTG we created a MySQL database to hold the Geonames dataset. It has a simple structure comprising a main table named “geoname” with one row per place, and a linked subsidiary table named “alternatename” that holds one row for each alternative name for a given place in the main table. There is also a smaller table named “hierarchy” that allows a hierarchical tree of places located within larger places to be constructed.

The database can be created by downloading the relevant files from the Geonames download server: “allCountries.zip”, “alt AttributeNames.zip” and “hierarchy.zip”. Once unzipped, these can be imported into a MySQL database - set the character encoding to UTF-8 when you create the database:

create database geonames character set utf8 collate utf8_general_ci;

You will need to set up suitable access permissions and will probably also want to create indexes to speed query performance.

We keep our copy of the database up to date by running nightly cron jobs to download and apply changes. To make this easy, an extra set of tables is used: “updates_geoname”, “updates_alternatename”, “deletes_geoname”, “deletes_alternatename”. The steps are:

1. Download the update files from the Geonames download server. These are named either “modifications” or “deletes” for the main table or the alternatename table, with a datestamp appended. Also download the hierarchy file.
2. Load the modification and deletion data into the four holding tables (clearing these of previous data first).
3. For the deletions, simply remove rows from “geoname” and “alternatename” that have a match in the holding tables for deletions.
4. For the modifications, remove matching rows from “geoname” and “alternatename” and then insert the rows from the holding tables.
5. Drop the hierarchy table then recreate and re-index it from the downloaded data.
6. Log the transactions carried out, for reporting.

If you want to create a local copy of geonames for yourself, there is a zip file of the database creation routine, daily update scripts, and cron file here (HTML documentation only). The directory names would need to be tailored to
your local setup. You may need to create a Geonames account name - see the Geonames website for details, as the policy seems to vary.

If a local copy of geonames is set up in this way then the -g geonames-local option can be used to access it with the geoparser; otherwise this option does not work. The gazlookup-geonames-local script must be edited to provide connection information for your local MySQL database. If the username is “pipeline”, with no password, then only the server location needs altering, as this is the default username in the script. The pipeline user should be set up as a read-only account, as the pipeline never alters data in the gazetteer. If the MySQL server is on the same machine as the pipeline is running, then the -h host parameter is not required. In this, the simplest case, the database connection string in the gazlookup-geonames-local scripts is:

```
lxmysql -u pipeline -d geonames
```

### 6.2.2 Example Setup: Pleiades+

In the same way that the “geonames-local” option only works if a local database is being maintained, -g plplus-local will work if and only if a local copy of Pleiades+ is created. This may be desirable because, at the time of writing, the Edina version is an out of date snapshot, and newer material is available as described above.

If a local version of Pleiades+ is created, then the relevant scripts included with the geoparser download bundle will be able to use it. In fact we have two versions: the one used for GAP and the newer version released in 2014. If you want to experiment with these, have a look at the gazlookup-plplus-OLDlocal (the GAP version) and gazlookup-plplus-NEWlocal scripts (the 2014 release). The -g plplus-local option is set to use the new version.

A bundle of scripts that may be helpful if you wish to set up your own local copy of the latest version of Pleiades and Pleiades+ is provided here (html documentation only). It includes routines for downloading the daily files and loading them into a database. These could easily be set up as cron jobs to refresh the database daily.

Note that the gazetteer lookup scripts to access a local pleiades database are at an experimental stage at present. The new database is much more complicated than that used by GAP and the queries take a bit longer, despite indexing. Also, there are a number of attributes provided by the full Pleiades dataset that could be used to refine the georesolution stage, but these alterations have not yet been attempted. As mentioned above, the LTG would welcome partners who would like to work with us on this.
Apart from Geonames, the online gazetteers referenced by the geoparser are all hosted by Edina’s Unlock service. There are two web services, Unlock Places and Unlock Text.

### 7.1 Unlock Places

Unlock Places provides an API to allow the user to match a placename string against one or more gazetteers. The geoparser uses this to find candidate locations for placename strings extracted by the geotagging step. The API provides a simple and flexible way to search different gazetteers, with results available in a range of formats.

For example:

```
http://unlock.edina.ac.uk/ws/search?name=Edinburgh
```

returns an xml file of candidate locations matching “Edinburgh” (108 candidates at time of writing) from all the available gazetteers, including whatever attributes the gazetteer provides - such as latitude, longitude, feature codes, population and alternative names.

To restrict the search to a particular gazetteer one simply specifies it in the request:

```
http://unlock.edina.ac.uk/ws/search?name=Halicarnassus&gazetteer=plplus
```

This returns matches for “Halicarnassus” in the copy of Pleiades+ hosted by Edina. (See Pleiades+ discussion in the Gazetteers chapter for more on versions of Pleiades.)

See the Unlock Places website for documentation and examples of use of the service.

### 7.2 Unlock Text

The Unlock Text API is a more sophisticated tool and requires the user to create an account. It allows you to submit complete texts for geoparsing, either individually or in bundles. It is, in effect, an online version of the geoparser pipeline and is probably the simplest way of using the geoparser, if no customisation is required. There are subtle and unavoidable differences between the Unlock version and the downloadable geoparser package, meaning that results will not necessarily be identical. If your needs are complex it may be better to install a local copy of the geoparser.

See Getting Started with Unlock Text for how to use the online service. At the time of writing, a visualisation component is in development, so that users can submit full texts and see map-based results immediately.

At the time of writing the Unlock Text documentation doesn’t explicitly show how to use the Pleiades+ gazetteer hosted by Edina, but this can be specified much as for the Unlock Text example above, by including a `gazetteer:gazname` pair in the JSON data item for “src”, like this:

```json
```
For more on using Unlock Text with classical works, see blog posts on the Google Ancient Places (GAP) project at Unlocking Text and Even More Unlocked. The GapVis interface (version 1 and version 2) uses Unlock Text as part of its back-end engine, to produce an interface to Herodotus’ *Histories* and other texts, for students of classics.
APPENDIX 1: LT-TTT2 TUTORIAL

The geotagging part of the pipeline is built using the Text Tokenisation Tool (LT-TTT2) developed by the Language Technology Group at Edinburgh. This in turn makes use of the LT-XML2 toolkit. Both LT-TTT2 and LT-XML2 are downloadable from the LTG software page.

There is a tutorial on the LTG website explaining how to use the LT-TTT2 suite, and in particular how to write and modify the grammars used by the lxtransduce program which is at the heart of LT-TTT2:

LT-TTT2 tutorial

This tutorial is also included in the documentation provided with the download of LT-TTT2.
APPENDIX 2: LTG PUBLICATIONS ABOUT THE GEOPARSER

This is a list of some research papers relating to the geoparser published by the Language Technology Group and our collaborators:


If citing the geoparser please use this one: