# Developing Language Processing Components with GATE (a User Guide)

For GATE version 2.0

Hamish Cunningham[1]
Diana Maynard[2]
Kalina Bontcheva[3]
Valentin Tablan[4]
Cristian Ursu[5]
Marin Dimitrov[6]



`http://gate.ac.uk/`

**HTML version:** `http://gate.ac.uk/sale/tao/`

This work has been supported by the Engineering and Physical Sciences Research Council (EPSRC) under grants GR/K25267 and GR/M31699, and by several smaller grants.

`$Id: tao_main.tex,v 1.22 2002/03/08 17:39:35 hamish Exp $`

[1] `http://www.dcs.shef.ac.uk/~hamish/`
[2] `http://www.dcs.shef.ac.uk/~diana/`
[3] `http://www.dcs.shef.ac.uk/~kalina/`
[4] `http://www.dcs.shef.ac.uk/~valyt/`
[5] `http://www.dcs.shef.ac.uk/~cursu/`
[6] `http://www.sirma.bg/marin.htm`

# Brief Contents

# Contents

# Chapter 1

# Introduction

> Software documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. (Anonmymous.)
>
> There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other way is to make it so complicated that there are no obvious deficiencies. (C.A.R. Hoare)
>
> A computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. (The Structure and Interpretation of Computer Programs, H. Abelson, G. Sussman and J. Sussman, 1985.)
>
> If you try to make something beautiful, it is often ugly. If you try to make something useful, it is often beautiful. (Oscar Wilde)[1]

GATE is an infrastructure for developing and deploying software components that process human language. GATE helps scientists and developers in three ways:

1. by specifiying an **architecture**, or organisational structure, for language processing software;
2. by providing a **framework**, or class library, that implements the architecture and can be used to embed language processing capabilities in diverse applications;
3. by providing a **development environment** built on top of the framework made up of convenient graphical tools for developing components.

The architecture exploits component-based software development, object orientation and mobile code. The framework and development environment are written in Java and

[1] These were, at least, our ideals; of course we didn't completely live up to them...

available as open-source free software under the GNU library licence[2]. GATE uses Unicode [The Unicode Consortium 96] throughout, and has been tested on a variety of Slavic, Germanic, Romance, and Indic languages [Maynard *et al.* 01, Gambäck & Olsson 00, McEnery *et al.* 00].

From a scientific point-of-view, GATE's contribution is to quantitative measurement of accuracy and repeatability of results for verification purposes.

GATE has been in development at the University of Sheffield since 1995 and has been used in a wide variety of research and development projects [Maynard *et al.* 00]. Version 1 of GATE was released in 1996, was licensed by several hundred organisations, and used in a wide range of language analysis contexts including Information Extraction ([Cunningham 99b, Appelt 99, Gaizauskas & Wilks 98, Cowie & Lehnert 96]) in English, Greek, Spanish, Swedish, German, Italian and French. Version 2 of the system, a complete re-implementation and extension of the original, is available from `http://gate.ac.uk/download/`.

This book describes how to use GATE to develop language processing components, test their performance and deploy them as parts of other applications. In the rest of this chapter:

- section 1.1 describes the best way to use this book;
- section 1.2 briefly notes that the context of GATE is applied language processing, or *Language Engineering*;
- section 1.3 gives an overview of developing using GATE;
- section 1.4 describes the structure of the rest of the book;
- section 1.5 lists other publications about GATE.

## 1.1 How to Use This Text

It is a good idea to read all of this introduction (you can skip sections 1.2 and 1.5 if pressed); then you can either continue wading through the whole thing or just use chapter 2 as a reference and dip into other chapters for more detail as necessary. Chapter 2 gives instructions for completing common tasks with GATE, organised in a FAQ style: details, and the reasoning behind the various aspects of the system, are omitted in this chapter, so where more information is needed refer to later chapters.

The structure of the book as a whole is detailed in section 1.4 below.

[2]This is a restricted form of the GNU licence, which means that GATE can be embedded in commercial products if required.

## 1.2 Context

GATE can be thought of as a **Software Architecture for Language Engineering**[3] [Cunningham 00].

'Software Architecture' is used rather loosely here to mean computer infrastructure for software development, including development environments and frameworks, as well as the more usual use of the term to denote a macro-level organisational structure for software systems [Shaw & Garlan 96].

Language Engineering (LE) may be defined as:

> ... the discipline or act of engineering software systems that perform tasks involving processing human language. Both the construction process and its outputs are measurable and predictable. The literature of the field relates to both application of relevant scientific results and a body of practice. [Cunningham 99a]

The relevant scientific results in this case are the outputs of Computational Linguistics, Natural Language Processing and Artificial Intelligence in general. Unlike these other disciplines, LE, as an engineering discipline, entails *predictability*, both of the process of constructing LE-based software and of the performance of that software after its completion and deployment in applications.

Some working definitions:

1. **Computational Linguistics (CL):** science of language that uses computation as an investigative tool.
2. **Natural Language Processing (NLP):** science of computation whose subject matter is data structures and algorithms for computer processing of human language.
3. **Language Engineering (LE):** building NLP systems whose cost and outputs are measurable and predictable.
4. **Software Architecture:** macro-level organisational principles for families of systems. In this context is also used as **infrastructure**.
5. **Software Architecture for Language Engineering (SALE):** software infrastructure, architecture and development tools for applied CL, NLP and LE.

(Of course the practice of these fields is broader and more complex than these definitions.)

In the scientific endeavours of NLP and CL, GATE's role is to support experimentation. In this context GATE's significant features include support for automated measurement (see

[3] `http://gate.ac.uk/sale/thesis/`

section 8), providing a 'level playing field' where results can easily be repeated across different sites and environments, and reducing research overheads in various ways.

## 1.3 Overview

### 1.3.1 Developing and Deploying Language Processing Facilities

GATE as an architecture suggests that the elements of software systems that process natural language can usefully be broken down into various types of component, known as resources[4]. Components are reusable software chunks with well-defined interfaces, and are a popular architectural form, used in Sun's Java Beans and Microsoft's .Net, for example. GATE components are specialised types of Java Bean, and come in three flavours:

- LanguageResources (LRs) represent entities such as lexicons, corpora or ontologies;
- ProcessingResources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers;
- VisualResources (VRs) represent visualisation and editing components that participate in GUIs.

These definitions can be blurred in practice as necessary.

Collectively, the set of resources integrated with GATE is known as **CREOLE**: a Collection of REusable Objects for Language Engineering. All the resources are packaged as Java Archive (or 'JAR') files, plus some XML configuration data. The JAR and XML files are made available to GATE by putting them on a web server, or simply placing them in the local file space. Section 1.3.2 introduces GATE's built-in resource set.

When using GATE to develop language processing functionality for an application, the developer uses the development environment and the framework to construct resources of the three types. This may involve programming, or the development of Language Resources such as grammars that are used by existing Processing Resources, or a mixture of both. The development environment is used for visualisation of the data structures produced and consumed during processing, and for debugging, performance measurement and so on. For example, figure 1.1 is a screenshot of one of the visualisation tools (displaying named-entity extraction results for a Bengali sentence).

[4]The terms 'resource' and 'component' are synonymous in this context. 'Resource' is used instead of just 'component' because it is a common term in the literature of the field: cf. the Language Resources and Evaluation conference series [LREC-1 98, LREC-2 00].

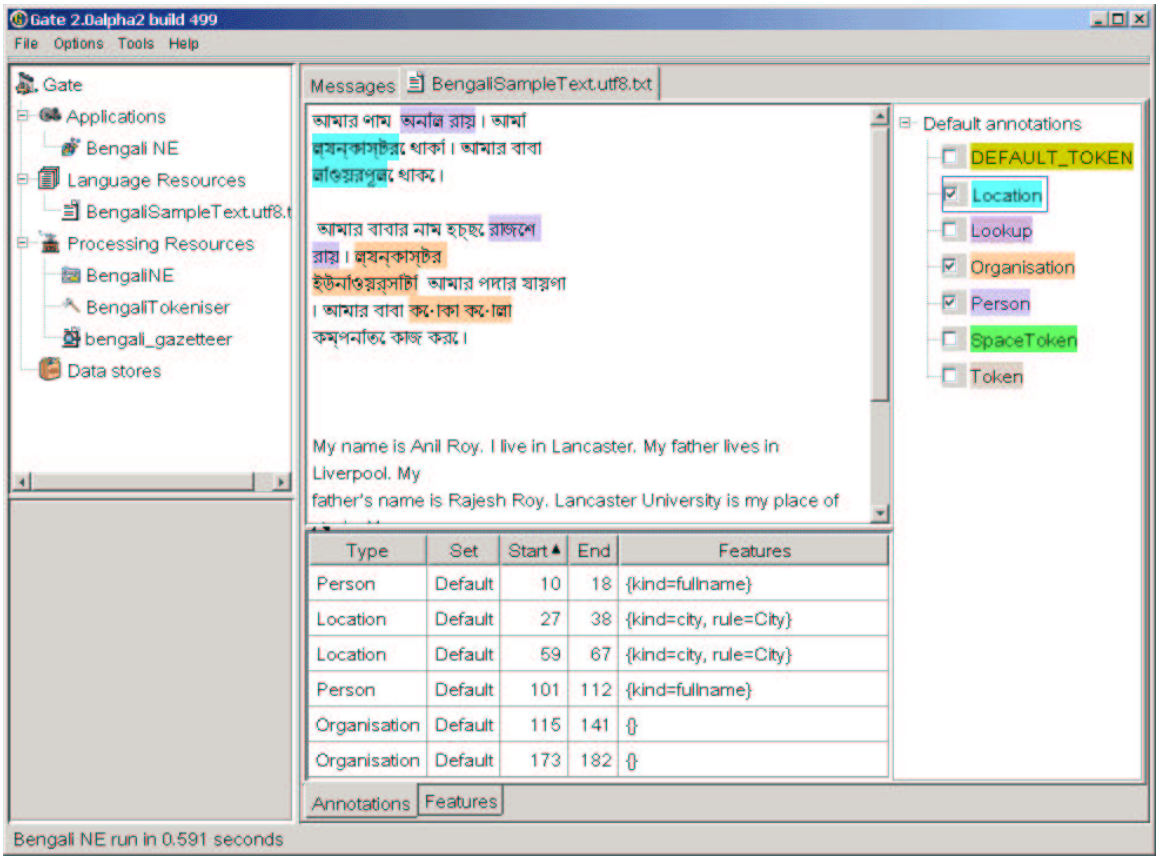



Figure 1.1: One of GATE's visual resources

The GATE development environment is analogous to systems like Mathematica for Mathematicians, or JBuilder for Java programmers: it provides a convenient graphical environment for research and development of language processing software.

When an appropriate set of resources have been developed, they can then be embedded in the target client application using the GATE framework. The framework is supplied as two JAR files.[5] To embed GATE-based language processing facilities in an application, these JAR files are all that is needed, along with JAR files and XML configuration files for the various resources that make up the new facilities.

### 1.3.2 Built-in Components

GATE includes resources for common LE data structures and algorithms, including documents, corpora and various annotation types, a set of language analysis components for Information Extraction and a range of data visualisation and editing components.

GATE supports documents in a variety of formats including XML, RTF, email, HTML, SGML and plain text. In all cases the format is analysed and converted into a single unified model of *annotation*. The annotation format is a modified form the TIPSTER format [Grishman 97] which has been made largely compatible with the Atlas format [Bird & Liberman 99], and uses the now standard mechanism of 'stand-off markup'. GATE documents, corpora and annotations are stored in databases of various sorts, visualised via the development environment, and accessed at code level via the framework. See chapter 4 for more details of corpora etc.

A family of Processing Resources for language analysis is included in the shape of ANNIE, A Nearly-New Information Extraction system. These components use finite state techniques to implement various tasks from tokenisation to semantic tagging or verb phrase chunking. All ANNIE components communicate exclusively via GATE's document and annotation resources. See chapter 6 for more details.

See chapter 7 for other miscellaneous CREOLE resources.

### 1.3.3 Additional Facilities

Three other facilities in GATE deserve special mention:

- JAPE, a Java Annotation Patterns Engine, provides regular-expression based pattern/action rules over annotations – see chapter 5.

[5] The main JAR file (**gate.jar**) supplies the framework, built-in resources and various 3rd-party libraries; the second file (**guk.jar**, the GATE Unicode Kit) contains Unicode support (e.g. additional input methods for languages not currently supported by the JDK). They are separate because the latter has to be a Java extension with a privileged security profile.

- The 'annotation diff' tool in the development environment implements performance metrics such as precision and recall for comparing annotations. Typically a language analysis component developer will mark up some documents by hand and then use these along with the diff tool to automatically measure the performance of the components. See section 8.
- GUK, the GATE Unicode Kit, fills in some of the gaps in the JDK's[6] support for Unicode, e.g. by adding input methods for various languages from Urdu to Chinese. See section 2.26 for more details.

And by version 3 it will make a mean cup of tea.

### 1.3.4 An Example

This section gives a very brief example of a typical use of GATE to develop and deploy language processing capabilities in an application, and to generate quantitative results for scientific publication.

Let's imagine that a developer called Fatima is building an email client[7] for Cyberdyne Systems' large corporate Intranet. In this application she would like to have a language processing system that automatically spots the names of people in the corporation and transforms them into `mailto` hyperlinks.

A little investigation shows that GATE's existing components can be tailored to this purpose. Fatima starts up the development environment, and creates a new document containing some example emails. She then loads some processing resources that will do named-entity recognition (a tokeniser, gazetteer and semantic tagger), and creates an application to run these components on the document in sequence. Having processed the emails, she can see the results in one of several viewers for annotations.

The GATE components are a decent start, but they need to be altered to deal specially with people from Cyberdyne's personnel database. Therefore Fatima creates new "cyber-" vesions of the gazetteer and semantic tagger resources, using the "bootstrap" tool. This tool creates a directory structure on disk that has some Java stub code, a Makefile and an XML configuration file. After several hours struggling with badly written documentation, Fatima manages to compile the stubs and create a JAR file containing the new resources. She tells GATE the URL of these files[8], and the system then allows her to load them in the same way that she loaded the built-in resources earlier on.

[6]JDK: Java Development Kit, Sun Microsystem's Java implementation. Unicode support is being actively improved by Sun, but at the time of writing many languages are still unsupported. In fact, Unicode itself doesn't support all languages, e.g. Sylheti; hopefully this will change in time.

[7]Perhaps because Outlook Express trashed her mail folder again, or because she got tired of Microsoft-specific viruses and hadn't heard of Netscape or Emacs.

[8]While developing, she uses a `file:/...` URL; for deployment she can put them on a web server.

Fatima then creates a second copy of the email document, and uses the annotation editing facilities to mark up the results that she would like to see her system producing. She saves this and the version that she ran GATE on into her Oracle datastore (set up for her by the Herculean efforts of the Cyberdyne technical support team, who like GATE because it enables them to claim lots of overtime). From now on she can follow this routine:

1. Run her application on the email test corpus.
2. Check the performance of the system by running the 'annotation diff' tool to compare her manual results with the system's results. This gives her both percentage accuracy figures and a graphical display of the differences between the machine and human outputs.
3. Make edits to the code, pattern grammars or gazetteer lists in her resources, and recompile where necessary.
4. Tell GATE to re-initialise the resources.
5. Go to 1.

To make the alterations that she requires, Fatima re-implements the ANNIE gazetteer so that it regenerates itself from the local personnel data. She then alters the pattern grammar in the semantic tagger to prioritise recognition of names from that source. This latter job involves learning the JAPE language (see chapter 5), but as this is based on regular expressions it isn't too difficult.

Eventually the system is running nicely, and her accuracy is 93% (there are still some problem cases, e.g. when people use nicknames, but the performance is good enough for production use). Now Fatima stops using the GATE development environment and works instead on embedding the new components in her email application. This application is written in Java, so embedding is very easy[9]: the two GATE JAR files are added to the project CLASSPATH, the new components are placed on a web server, and with a little code to do initialisation, loading of components and so on, the job is finished in half a day – the code to talk to GATE takes up only around 150 lines of the eventual application, most of which is just copied from the example in the `sheffield.examples.StandAloneAnnie`[10] class.

Because Fatima is worried about Cyberdyne's unethical policy of developing Skynet to help the large corporates of the West strengthen their strangle-hold over the World, she wants to get a job as an academic instead (so that her conscience will only have to cope with the torture of students, as opposed to humanity). She takes the accuracy measures that she has attained for her system and writes a paper for the Journal of Nasturtium Logarithm Encitement describing the approach used and the results obtained. Because she used GATE

[9]Languages other than Java require an additional interface layer, such as JNI, the Java Native Interface, which is in C.

[10]`http://gate.ac.uk/GateExamples/doc/java2html/sheffield/examples/StandAloneAnnie.java.html`

for development, she can cite the repeatability of her experiments and offer access to example binary versions of her software by putting them on an external web server.

And everybody lived happily ever after.

## 1.4 Structure of the Book

The material presented in this book ranges from the conceptual (e.g. 'what is software architecture?') to practical instructions for programmers (e.g. how to deal with GATE exceptions) and linguists (e.g. how to write a pattern grammar). This diversity is something of an organisational challenge. Our (no doubt imperfect) solution is to collect specific instructions for 'how to do X' in a separate chapter (2). Other chapters give a more discursive presentation. In order to understand the whole system you must, unfortunately, read much of the book; in order to get help with a particular task, however, look first in chapter 2 and refer to other material as necessary.

The other chapters:

Chapter 3 describes the GATE architecture's component-based model of language processing, describes the lifecycle of GATE components, and how they can be grouped into applications and stored in databases and files.

Chapter 4 describes GATE's model of document formats, annotated documents, annotation types, and corpora (sets of documents). It also covers GATE's facilities for reading and writing in the XML data interchange language.

Chapter 5 describes JAPE, a pattern/action rule language based on regular expressions over annotations on documents. JAPE grammars compile into cascaded finite state transducers.

Chapter 6 describes ANNIE, a pipelined Information Extraction system which is supplied with GATE.

Chapter 7 describes CREOLE resources bundled with the system that don't fit into the previous categories.

Chapter 8 describes how to measure the performance of language analysis components.

Chapter 9 describes the data store security model.

Appendix A discusses the design of the system.

Appendix B describes the implementation details and formal definitions of the JAPE annotation patterns language.

Appendix C describes in some detail the JAPE pattern grammars that are used in ANNIE for named-entity recognition.

## 1.5 Further Reading

Lots of documentation lives on the GATE web server[11], including:

- the main system documentation tree[12];
- JavaDoc API documentation[13];
- HTML of the source code[14];
- parts of the requirements analysis[15] that version 2 is based on.

For more details about Sheffield University's work in human language processing see the NLP group pages[16] or [Cunningham 99a][17]. For more details about Information Extraction see *IE, a User Guide*[18] or the Sheffield IE pages[19].

A list of publications on GATE and projects that use it (some of which are available online[20]):

[**Maynard** *et al.* **02**] describes robustness and predictability in LE systems, and presents GATE as an example of a system which contributes to robustness and to low overhead systems development.

[**Maynard** *et al.* **01**] discusses a project using ANNIE for named-entity recognition across wide varieties of text type and genre.

[**Cunningham 00**] defines the field of Software Architecture for Language Engineering, reviews previous work in the area, presents a requirements analysis for such systems (which was used as the basis for designing GATE version 2), and evaluates the strengths and weaknesses of GATE version 1.

[**Cunningham 02**] describes the philosophy and motivation behind the system, describes GATE version 1 and how well it lived up to its design brief.

[11] `http://gate.ac.uk/`
[12] `http://gate.ac.uk/gate/doc/`
[13] `http://gate.ac.uk/gate/doc/javadoc`
[14] `http://gate.ac.uk/gate/doc/java2html`
[15] `http://gate.ac.uk/gate/doc/usecases.html`
[16] `http://nlp.shef.ac.uk/`
[17] `http://www.dcs.shef.ac.uk/~hamish/LeIntro.html`
[18] `http://www.dcs.shef.ac.uk/~hamish/IE/`
[19] `http://www.dcs.shef.ac.uk/nlp/extraction`
[20] `http://gate.ac.uk/gate/doc/papers.html`

[**McEnery** *et al.* **00**] presents the EMILLE project[21] in the context of which GATE's Unicode support for Indic languages has been developed.

[**Cunningham** *et al.* **00d**] **and** [**Cunningham 99c**] document early versions of JAPE (superceded by the present document).

[**Cunningham** *et al.* **00a**], [**Cunningham** *et al.* **98a**] **and** [**Peters** *et al.* **98**] presents GATE's model of Language Resources, their access and distribution.

[**Maynard** *et al.* **00**] surveys users of GATE up to mid-2000.

[**Cunningham** *et al.* **00c**] **and** [**Cunningham** *et al.* **99**] summarise experiences with GATE version 1.

[**Cunningham** *et al.* **00b**] taxonomises Language Engineering components and discusses the requirements analysis for GATE version 2.

[**Bontcheva** *et al.* **00**] **and** [**Brugman** *et al.* **99**] describe a prototype of GATE version 2 that integrated with the EUDICO multimedia markup tool[22] from the Max Planck Institute.

[**Gambäck & Olsson 00**] discusses experiences in the Svensk project, which used GATE version 1 to develop a reusable toolbox of Swedish language processing components.

[**Stevenson** *et al.* **98**] **and** [**Cunningham** *et al.* **98b**] report work on implementing a word sense tagger in GATE version 1.

[**Cunningham** *et al.* **97b**] presents motivation for GATE and GATE-like infrastructural systems for Language Engineering.

[**Gaizauskas** *et al.* **96b, Cunningham** *et al.* **97a, Cunningham** *et al.* **96e**] report work on GATE version 1.

[**Cunningham** *et al.* **96c, Cunningham** *et al.* **96d, Cunningham** *et al.* **95**] report early work on GATE version 1.

[**Cunningham** *et al.* **96b**] discusses a selection of projects in Sheffield using GATE version 1 and the TIPSTER architecture it implemented.

[**Cunningham** *et al.* **96a**] was the guide to developing CREOLE components for GATE version 1.

[**Gaizauskas** *et al.* **96a**] was the user guide for GATE version 1.

[**Humphreys** *et al.* **96**] desribes the language processing components distributed with GATE version 1.

[21] `http://www.emille.lancs.ac.uk/`

[22] `http://www.mpi.nl/world/tg/lapp/eudico/eudico.html`

[**Cunningham 94, Cunningham** *et al.* **94**] argue that software engineering issues such as reuse, and framework construction, are important for language processing R&D.

Never in the history of the Research Assessment Exercise has so much been owed by so many to so few exercises in copy-and-paste.

# Chapter 2

# How To...

> "The law of evolution is that the strongest survives!"
>
> "Yes; and the strongest, in the existence of any social species, are those who are most social. In human terms, most ethical. ... There is no strength to be gained from hurting one another. Only weakness."
>
> The Dispossessed [p.183], Ursula K. le Guin, 1974.

This chapter describes how to complete common tasks using GATE. Sections that relate to the Development Environment are flagged [**D**]; those that relate to the framework are flagged [**F**]; sections relating to both are flagged [**D,F**].

There are two other primary sources for this type of information:

- for the development enviroment, see the visual tutorials available on our 'movies' page[1];
- for the framework, see the example code at `http://gate.ac.uk/GateExamples/doc/`.

## 2.1 Download GATE

To download GATE point your web browser at `http://gate.ac.uk/` and follow the download link. Fill in the form there, and you will be emailed an FTP address to download the system from.

[1] `http://gate.ac.uk/demos/movies.html`

## 2.2 Install and Run GATE

GATE will run anywhere that supports recent versions of Java, including Solaris, Linux and Windoze platforms. We don't run tests on other platforms, but have had reports of successfull installs elsewhere (e.g. MacOS X).

### 2.2.1 The Easy Way

The easy way to install is to use one of the platform-specific installers (created using ZeroG's InstallAnywhere[2] product). Download a 'platform-specific installer' and follow the instructions it gives you.

### 2.2.2 The Hard Way

Download one of the Java-only release packages, and follow the instructions below.

**Prerequisites:**

- A conforming Java 2 environment, version 1.3 or above, available free from Sun Microsystems[3] or from your UNIX supplier. (We test on various Sun 1.3 and 1.4 JDKs on Solaris, Linux, NT 4, Windoze 2000 and Windoze XP.)
- Binaries from the GATE distribution you downloaded: `gate.jar`, `guk.jar` (Unicode editing support) and a suitable script to start Java, e.g. `gate.sh` or `gate.bat`. These are held in a directory called `bin` like this:

```
.../bin/
    gate.jar
    gate.sh
    gate.bat
    .../bin/ext/
        guk.jar
```

- An open mind and a sense of humour.

Using the binary distribution:

[2] `http://www.zerog.com/`
[3] `http://java.sun.com/products/jdk/`

- Unpack the distribution, creating a directory containing jar files and scripts.
- If you want to copy the scripts that run the system somewhere else, then you need to set some environment variables – see below.
- To run the development environment: on Windows, click on `gate.bat`; on UNIX run `gate.sh`.
- To embed GATE as a library, put `gate.jar` in your `CLASSPATH` and tell Java that guk.jar is an extension (`-Djava.ext.dirs=path-to-guk.jar`).

The scripts that start GATE (`gate.bat` or `gate.sh`) use three environment variables:

1. `JAVA_HOME` should point to the location of `java`; if it is not set the scripts assume that it is in your `PATH`. (Note: the ZeroG installer sets this variable for you in the startup script.)
2. `GATE_HOME` should point to the directory containing the binaries directory `bin` (which contains `gate.jar`, and the `ext` directory containing `guk.jar`). If not set, the scripts assume that these files live in the same directory as the script, or in `../build` and `../lib/ext` (so that you can use the script for full source distributions of the system as well as the binary distributions). (Note: the ZeroG installer sets this variable for you in the startup script.)
3. `GATE_CONFIG` should point to the directory containing the site-wide `gate.xml` configuration file (if such is required) – see section 2.5. If this is not set the scripts check the directory found for item 2 for any `gate.xml` file that may be present there.

The value of `GATE_CONFIG` is passed to the system by the scripts using either a `-i` command-line option, or the Java property `gate.config`.

## 2.3 Troubelshooting

Note: on Windoze 95 and 98, you may need to increase the amount of environment space available for the `gate.bat` script. Right click on the script, hit the memory tab and increase the 'initial environment' value to maximum.

## 2.4 [D] Get Started with the GUI

Probably the best way to learn how to use the GATE graphical development environment is to look at the animated demonstrations and tutorials on the 'movies' page[4]. This section

[4] `http://gate.ac.uk/demos/movies.html`

gives a short description of what is where in the main window of the system.

Figure 2.1 shows the main window of the application, with a single document loaded. There

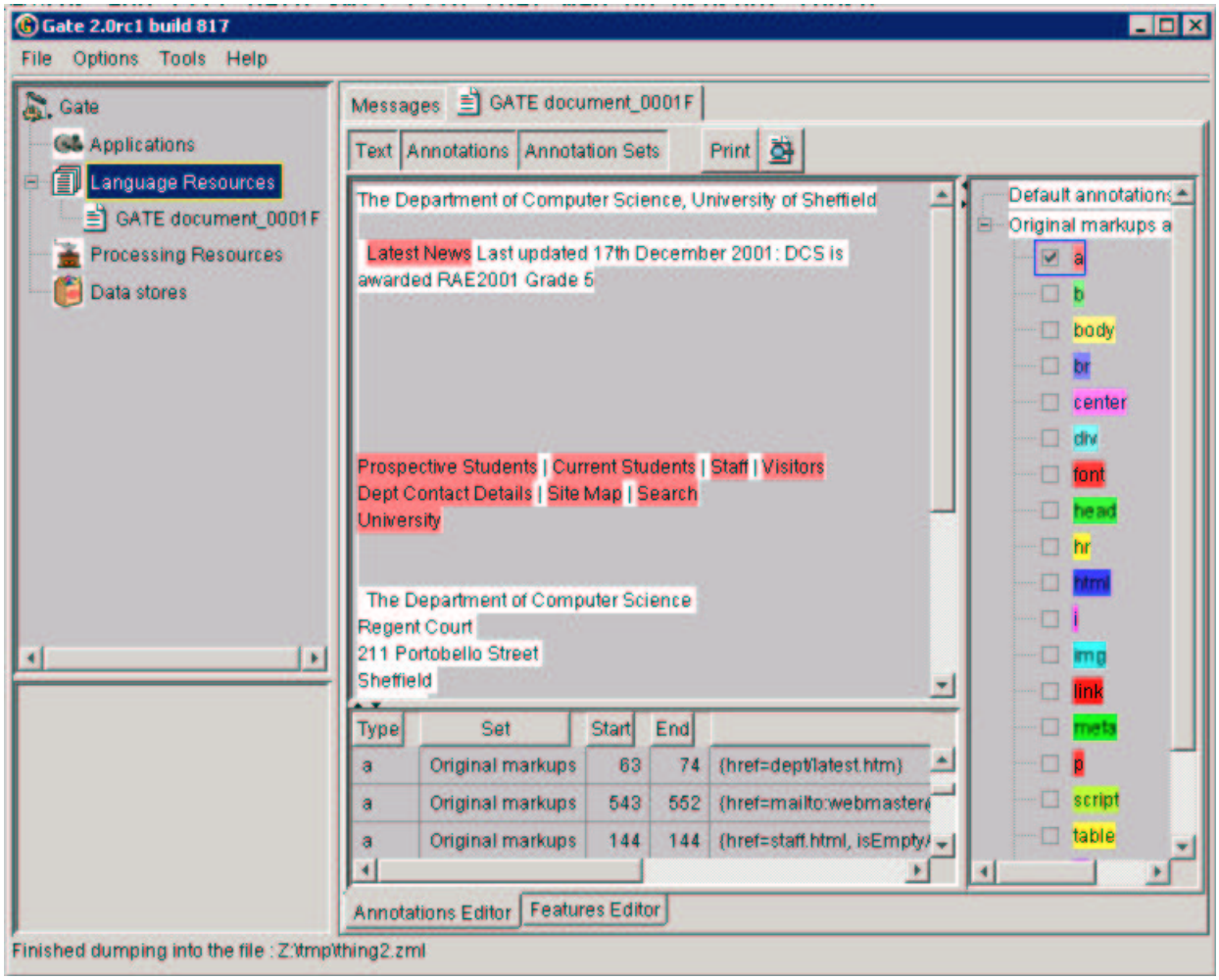



Figure 2.1: Main Window

are five main areas of the window:

1. the *menus bar* along the top, with 'File' etc.;
2. in the top left of the main area, a tree starting from 'Gate' and containing 'Applications', 'Language Resources' etc. – this is the *resources tree*;
3. in the bottom left of the main area, a black rectangle, which is the *small resource viewer*;
4. on the right of the main area, containing tabs with 'Messages' and 'GATE Document_0001F', the *main resource viewer*;
5. the *messages bar* along the bottom (where it says 'Finished dumping...').

The menu and the messages bars do the usual things. Longer messages are displayed in the messages tab in the main resource viewer area.

The resource tree and resource viewer areas work together to allow the system to display diverse resources in various ways. Visual Resources integrated with GATE can have a small view or a large view. For example, data stores have a small view; documents have a large view.

All the resources, applications and datastores currently loaded in the system appear in the resources tree; double clicking on a resource will load a viewer for the resource in one of the resource view areas.

## 2.5 [D,F] Configure GATE

When the GATE development environment is started, or when `Gate.init()` is called from the API, GATE loads various sorts of configuration data stored as XML in files generally called something like `gate.xml` or `.gate.xml`. This data holds information such as:

- whether to save settings on exit;
- what fonts the GUI should use;
- where the local Oracle database lives.

All of this type of data is stored at three levels (in order from general to specific):

- the system-wide level, located in the GATE resources packaged with the system[5];
- the site-wide level, whose location is specified by the environment variable `GATE_CONFIG` or the Java property `gate.config`;
- the user level, which lives in the user's HOME directory on UNIX or their profile directory on Windoze (note that parts of this file are overwritten by GATE when saving user settings).

Where configuration data appears on several different levels, the more specific ones overwrite the more general. This means that you can set defaults for all GATE users on your system, for example, and allow individual users to override those defaults without interfering with others.

[5]These are either in `gate/classes` or in `gate.jar` depending on whether you have a binary or a built version.

When using the GATE development environment, the side-wide config file is chosen depending on the settings of `GATE_CONFIG` and `GATE_HOME` (see section 2.2).

Configuration data can be set from the GUI via the 'Options' menu, 'Configuration' choice. The user can change the appearance of the GUI (via the Appearance submenu), which includes the options of font and the "look and feel". The "Advanced" submenu enables the user to include annotation features when saving the document and preserving its format, to save the selected Options automatically on exit, and to save the session automatically on exit. The Input Methods menu (available via the Options menu) enables the user to change the default language for input. These options are all stored in the user's .gate.xml file.

When using GATE from the framework, you can also set the site config location using `Gate.setSiteConfigFile(File)` prior to calling `Gate.init()`.

When using GATE from the framework, you can also set the site config location using `Gate.setSiteConfigFile(File)` prior to calling `Gate.init()`.

### 2.5.1 [F] Save Config Data to gate.xml

Arbitrary feature/value data items can be saved to the user's `gate.xml` file via the following API calls:

To get the config data: `Map configData = Gate.getUserConfig()`.

To add config data simply put pairs into the map: `configData.put("my new config key", "value");`.

To write the config data back to the XML file: `Gate.writeUserConfig();`.

Note that new config data will simply override old values, where the keys are the same. In this way defaults can be set up by putting their values in the main gate.xml file, or the site gate.xml file; they can then be overridden by the user's gate.xml file.

## 2.6 Build GATE

Note that you don't need to build GATE unless you're doing development on the system itself.

**Prerequisites:**

- A conforming Java environment as above.

- The UNIX shell tools, `make` etc. On Windoze use the excellent Cygwin distribution from Cygnus/Red Hat, available free from:
  - Cygwin.com[6]
  - The UK mirrors facility (HTTP)[7]
  - The UK mirrors facility (FTP)[8]

  If you're running on UNIX you've got this stuff already (though you may need to install GNU `make`, available at `http://www.gnu.org/`): pat yourself on the back and sigh contentedly in the knowledge that your 1970s operating system is still the best available.
- An appreciation of natural beauty.

**To build gate**, cd to gate/build and:

1. Start a shell (if you're on Windoze you can use the cygnus.bat script from the Cygwin distribution to do this). Make sure GNU `make` is in your PATH.
2. Run the script `configure`. Ignore any insulting messages.
3. Check that Makefile has the right paths for programs like `java` and `javac`. If configure made a mistake edit the paths.
4. Construct the dependency list and build the class files and a jar file containing the whole thing:
   ```
   make depend
   make
   make jar
   ```
5. [optional] To test the system, and build the Javadoc code documentation:
   ```
   make test
   ```
   (Note that DB tests may fail unless you can connect to Sheffield's Oracle server.)
   ```
   make docs
   ```
   (Note that this needs an open network connection.)
6. [optional] To build the developer Javadoc documentation (including private members):
   ```
   make internaldocs
   ```
   (Note that you may need to change the name of the server for the Java platform documentation in the Makefile, as we keep a copy on an internal server in Sheffield.)

[6] `http://www.cygwin.com`
[7] `http://www.mirror.ac.uk/sites/sourceware.cygnus.com/pub/cygwin/`
[8] `ftp://www.mirror.ac.uk`

(Those of you who used GATE 1 and who miss the baroque mystery and labyrinthine elegance of the previous build structure may find it useful to note that drinking milk can ease the discomfort of an acid stomach.)

You can also use a development environment like Borland JBuilder (click on the `gate.jpx` file), but note that it's still advisable to use the Makefile to generate documentation, the jar file and so on. Also note that the run configurations have the location of a `gate.xml` site configuration file hard-coded into them, so you may need to change these for your site.

## 2.7 [D,F] Create a New CREOLE Resource

CREOLE resources are Java Beans (see chapter 3). They come in three types: Language Resource, Processing Resource and Visual Resource (see chapter 1 section 1.3.1). To create a new resource you need to:

- write a Java class that implements GATE's beans model;
- compile the class, and any others that it uses, into a Java Archive (JAR) file;
- write some XML configuration data for the new resource;
- tell GATE the URL of the new JAR and XML files.

The GATE development environment helps you with this process by creating a set of directories and files that implement a basic resource, including a Java code file and a Makefile. This process is called 'bootstrapping'.

For example, let's create a new component called GoldFish, which will be a Processing Resource that looks for all instances of the word 'fish' in a document and adds an annotation of type 'GoldFish'.

First start the GATE development environment (see section 2.2). From the 'Tools' menu select 'BootStrap Wizard', which will pop up the dialogue in figure 2.2. The meaning of the data entry fields:

- The 'resource name' will be displayed when GATE loads the resource, and will be the name of the directory the resource lives in. For our example: `GoldFish`.

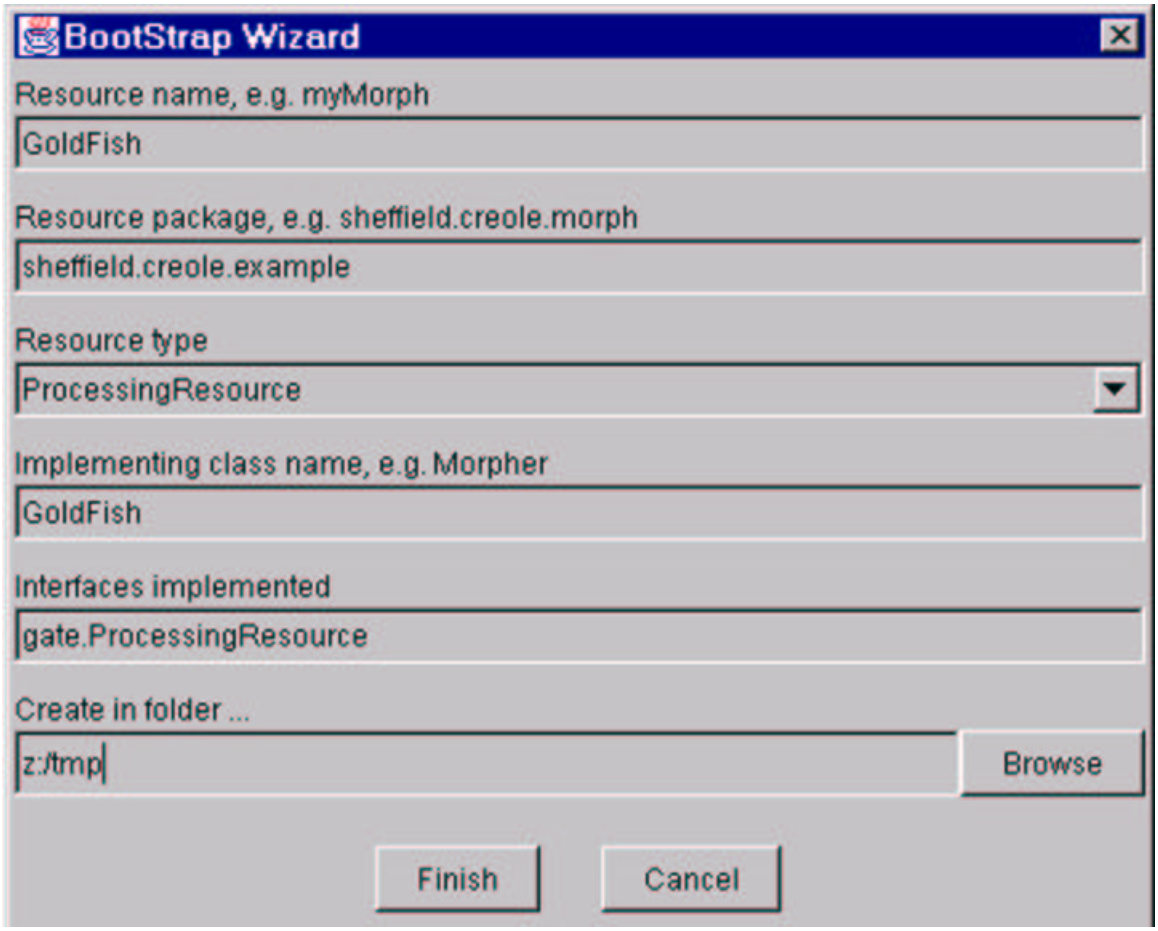



Figure 2.2: BootStrap Wizard Dialogue

- 'Resource package' is the Java package that the class representing the resource will be created in. For our example: `sheffield.creole.example`.
- 'Resource type' must be one of Language, Processing or Visual Resource. In this case we're going to process documents (and add annotations to them), so we select `ProcessingResource`.
- 'Implementing class name' is the name of the Java class that represents the resource. For our example: `GoldFish`.
- The 'interfaces implemented' field allows you to add other interfaces (e.g. `java.util.Set`) that you would like your new resource to implmenent. In this case we just leave the default (which is to implement the `gate.ProcessingResource` interface).
- The last field selects the directory that you want the new resource created in. For our example: `z:/tmp`.

Now we need to compile the class, and create the JAR and XML files that allow GATE to load the new resource. (There's no reason not to use your own favourite alternative, e.g.

ANT[9].) For the pre-requisites of the build process that we use (based on Makefiles, and the GNU shell tools) see section 2.6. When you have these pre-requisites available do the following from a command prompt (working from the `GoldFish/build` directory that the bootstrapper created for you):

```
./configure
make depend
make
make jar
```

This will create the two files that GATE needs to load your new resource: `GoldFish.jar` and `creole.xml`.

You can now load this resource into GATE; see

- section 2.8 for how to instantiate the resource from the framework;
- section 2.9 for how to load the resource in the development environment;
- section 2.10 for how to configure and further develop your resource (which will, by default, do nothing!).

The default Java code that was created for our GoldFish resource looks like this:

```
/*
 *  GoldFish.java
 *
 *  You should probably put a copyright notice here. Why not use the
 *  GNU licence? (See http://www.gnu.org/.)
 *
 *  hamish, 26/9/2001
 *
 *  $Id: howto.tex,v 1.63 2002/03/13 16:11:23 hamish Exp $
 */

package sheffield.creole.example;

import java.util.*;
import gate.*;
import gate.creole.*;
import gate.util.*;

/**
 * This class is the implementation of the resource GOLDFISH.
 */
public class GoldFish extends AbstractProcessingResource
```

[9] `http://jakarta.apache.org/ant/`

```
  implements ProcessingResource {


} // class GoldFish
```

The default XML configuration for GoldFish looks like this:

```
<!-- resource.xml GoldFish -->
<!--  hamish, 26/9/2001 -->
<!-- $Id: howto.tex,v 1.63 2002/03/13 16:11:23 hamish Exp $ -->

<CREOLE-DIRECTORY>

<CREOLE>
  <RESOURCE>
    <NAME>GoldFish</NAME>
    <JAR>GoldFish.jar</JAR>
    <CLASS>sheffield.creole.example.GoldFish</CLASS>
  </RESOURCE>
</CREOLE>

</CREOLE-DIRECTORY>
```

The directory structure containing these files is shown in figure 2.3. `GoldFish.java` lives


Figure 2.3: BootStrap directory tree

in the `src/sheffield/creole/example` directory. `creole.xml` is generated in the `build` directory from a source file called `resource.xml` which lives in the `src` directory. (The `lib`

directory is for libraries; the `classes` directory is where Java class files are placed; the `doc` directory is for documentation.)

This process has the advantage that it creates a complete source tree and build structure for the component, and the disadvantage that it creates a complete source tree and build structure for the component. If you already have a source tree, you will need to chop out the bits you need from the new tree (in this case `GoldFish.java` and `resource.xml`) and copy it into your existing one.

## 2.8 [F] Instantiate CREOLE Resources

This section describes how to create CREOLE resources as objects in a running Java virtual machine. This process involves using GATE's `Factory` class, and, in the case of LRs, may also involve using a `DataStore`.

CREOLE resources are Java Beans; creation of a resource object involves using a default constructor, then setting parameters on the bean, then calling an `init()` method[10]. The Factory takes care of all this, makes sure that the GUI is told about what is happenning (when GUI components exist at runtime), and also takes care of restoring LRs from DataStores. So a programmer using GATE should **never call the constructor** of a resource: always use the Factory.

The valid parameters for a resource are described in the resource's section of its `creole.xml` file – see section 2.10.

Creating a resource via the Factory involves passing values for any create-time parameters that require setting to the Factory's `createResource` method. If no parameters are passed, the defaults are used. So, for example, the following code creates a default ANNIE part-of-speech tagger:

```
FeatureMap params = Factory.newFeatureMap(); // empty map: default parameters
ProcessingResource tagger = (ProcessingResource)
  Factory.createResource("gate.creole.POSTagger", params);
```

Note that if the resource created here had any parameters that were both mandatory and had no default value, the `createResource` call would throw an exception. In this case, all the information needed to create a tagger is available in default values given in the tagger's XML definition:

```
<RESOURCE>
  <NAME>ANNIE POS Tagger</NAME>
```

[10]This method is not part of the beans spec.

```
  <COMMENT>Mark Hepple's Brill-style POS tagger</COMMENT>
  <CLASS>gate.creole.POSTagger</CLASS>
  <PARAMETER NAME="document"
    COMMENT="The document to be processed"
    RUNTIME="true">gate.Document</PARAMETER>
....
  <PARAMETER NAME="rulesURL" DEFAULT="gate:/creole/heptag/ruleset"
    COMMENT="The URL for the ruleset file"
    OPTIONAL="true">java.net.URL</PARAMETER>
</RESOURCE>
```

Here the two parameters shown are either 'runtime' parameters, which are set before a PR is executed, or have a default value (in this case the default rules file is distributed with GATE itself).

When creating a Document, however, the URL of the source for the document must be provided[11]. For example:

```
URL u = new URL("http://gate.ac.uk/hamish/");
FeatureMap params = Factory.newFeatureMap();
params.put("sourceUrl", u);
Document doc = (Document)
  Factory.createResource("gate.corpora.DocumentImpl", params);
```

The document created here is transient: when you quit the JVM the document will no longer exist. If you want the document to be persistent, you need to store it in a `DataStore`. Assuming that you have a `DataStore` already open called `myDataStore`, this code will ask the data store to take over persistence of your document, and to synchronise the memory representation of the document with the disk storage:

```
Document persistentDoc = myDataStore.adopt(doc, mySecurity);
myDataStore.sync(persistentDoc);
```

**Security:**
User access to the LRs is provided by a security mechanism of users and groups, similar to those on an operating system. When users create/save LRs into Oracle, they specify reading and writing access rights for users from their group and other users. For example, LRs created by one user/group can be made read-only to others, so they can use the data, but not modify it. The access modes are:

- others: read/none;

[11] Alternatively a string giving the document source may be provided.

- group: modify/read/none;
- owner: modify/read.

If needed, ownership can be transferred from one user to another. Users, groups and LR permissions are administered in a special administration tool, by a privileged user. For more details see chapter 9.

When you want to restore a document (or other LR) from a data store, you make the same `createResource` call to the Factory as for the creation of a transient resource, but this time you tell it the data store the resource came from, and the ID of the resource in that datastore:

```
KALI: CODE FOR RESTORING A DOC?
```

## 2.9 [D] Load CREOLE Resources

### 2.9.1 Loading Language Resources

Load a language resource by right clicking on "Language Resources" and selecting a language resource type (document, corpus or annotation schema). Choose a name for the resource, and choose any parameters as necessary.

For a document, a file or url should be selected as the value of "sourceUrl" (double clicking in the "values" box brings up a tree structure to enable selection of documents). Other parameters can be selected or changed as necessary, such as the encoding of the document, and whether it should be markup aware.

There are three ways of adding documents to a corpus:

1. When creating the corpus, clicking on the icon under Value brings up a popup window with a list of the documents already loaded into Gate. This enables the user to add any documents to the corpus.
2. Alternatively, the corpus can be loaded first, and documents added later by double clicking on the corpus and using the + and - icons to add or remove documents to the corpus. Note that the documents must have been loaded into Gate before they can be added to the corpus.
3. Once loaded, the corpus can be populated by right clicking on the corpus and selecting "Populate". With this method, documents do not have to have been previously loaded into Gate, as they will be loaded during the population process. Select the directory containing the relevant files, choose the encoding, and check or uncheck the "recurse directories" box as appropriate. The initial value for the encoding is the platform default.

To add a new annotation schema, simply choose the name and the path or Url. For more information about schema, see 4.4.1.

### 2.9.2 Loading Processing Resources

This section describes how to load and run CREOLE resources not present in ANNIE. To load ANNIE, see Section 2.13. For technical descriptions of these resources, see Chapter 7. All these resources are loaded by selecting them from the set of Processing Resources (right click on Processing Resources or select "New Processing Resource" from the File menu), adding them to the application and selecting the input and output Annotation Sets (and any other parameters as necessary).

#### Flexible Exporter

At load time, the following parameters can be set for the flexible exporter:

- includeFeatures - if set to true, features are included with the annotations exported; if false (the default status), they are not.
- useSuffixForDumpFiles - if set to true (the default status), the output files have the suffix defined in suffixForDumpFiles; if false, no suffix is defined, and the output file simply overwrites the existing file (but see the outputFileUrl runtime parameter for an alternative).
- suffixForDumpFiles - this defines the suffix if useSuffixForDumpFiles is set to true. By default the suffix is .gate.

The following runtime parameters can also be set (after the file has been selected for the application):

- annotationSetName - this enables the user to specify the name of the annotation set which contains the annotations to be exported. If no annotation set is defined, it will use the Default annotation set.
- annotationTypes - this contains a list of the annotations to be exported. By default it is set to Person, Location and Date.
- dumpTypes - this contains a list of names for the exported annotations. If the annotation name is to remain the same, this list should be identical to the list in annotationTypes. The list of annotation names must be in the same order as the corresponding annotation types in annotationTypes.
- outputfileUrl - this enables the user to select a different name for the output file. The file will be stored in the same directory as the original source file.

**Annotation Set Transfer**

The Annotation Set Transfer has no loadtime parameters. It has the following runtime parameters:

- inputASName - this defines the annotation set which is to be transferred. If nothing is specified, the Default annotation set will be used.
- outputASName - this defines the new annotation set which will contain the transferred annotations. The default for this is a set called Filtered.
- tagASName - this defines the annotation set which contains the annotation covering the relevant part of the document to be transferred.
- textTagName - this defines the name of the annotation covering the relevant part of the document to be transferred.

For example, suppose we wish to perform named entity recognition on only the text covered by the BODY annotation from the Original Markups annotation set in an HTML document. We have to run the gazetteer and tokeniser on the entire document, because since these resources do not depend on any other annotations, we cannot specify an input annotation set for them to use. We therefore transfer these annotations to a new annotation set (Filtered) and then perform the NE recognition over these annotations, by specifying this annotation set as the input annotation set for all the following resources. In this example, we would set the following parameters (assuming that the annotations from the tokenise and gazetteer are initially placed in the Default annotation set).

- inputASName: Default
- outputASName: Filtered
- tagASName: Original markups
- textTagName: BODY

## 2.10 [D,F] Configure CREOLE Resources

This section desscribes how to write entries in the `creole.xml` file that is used to describe resources to GATE. This data is used to tell GATE things like what parameters a resource has, how to display it if it has a visualisation, etc.

An example file:

```
<CREOLE-DIRECTORY>
  <CREOLE>
    <RESOURCE>
      <NAME>GATE XML Document Format</NAME>
      <CLASS>gate.corpora.XmlDocumentFormat</CLASS>
      <AUTOINSTANCE/>
      <PRIVATE/>
      <JAR>gate.jar</JAR>
    </RESOURCE>
  </CREOLE>
</CREOLE-DIRECTORY>
```

These files have as a root element `CREOLE-DIRECTORY`, and may contain any number of `CREOLE` elements, which in turn contain any number of `RESOURCE` elements[12].

Each resource must give a name, a Java class and the JAR file that it can be loaded from. The above example defines GATE's XML document format analyser resource. This resource has no parameters, is automatically loaded when the `creole.xml` data is loaded, is not displayed to the GUI user (it is used internally by the document creation code), and is loaded from `gate.jar`.

Resources may also have parameters of various types. These resources, from the GATE distribution, illustrate the various types of parameters:

```
<RESOURCE>
  <NAME>GATE document</NAME>
  <CLASS>gate.corpora.DocumentImpl</CLASS>
  <INTERFACE>gate.Document</INTERFACE>
  <COMMENT>GATE transient document</COMMENT>
  <OR>
    <PARAMETER NAME="sourceUrl"
      SUFFIXES="txt;text;xml;xhtm;xhtml;html;htm;sgml;sgm;mail;email;eml;rtf"
      COMMENT="Source URL">java.net.URL</PARAMETER>
    <PARAMETER NAME="stringContent"
      COMMENT="The content of the document">java.lang.String</PARAMETER>
  </OR>
  <PARAMETER
    COMMENT="Should the document read the original markup"
    NAME="markupAware" DEFAULT="true">java.lang.Boolean</PARAMETER>
  <PARAMETER NAME="encoding" OPTIONAL="true"
```

[12]The purpose of the `CREOLE` element is to allow files to be build up from the concatenation of multiple other files.

```
    COMMENT="Encoding" DEFAULT="">java.lang.String</PARAMETER>
  <PARAMETER NAME="sourceUrlStartOffset"
    COMMENT="Start offset for documents based on ranges"
    OPTIONAL="true">java.lang.Long</PARAMETER>
  <PARAMETER NAME="sourceUrlEndOffset"
    COMMENT="End offset for documents based on ranges"
    OPTIONAL="true">java.lang.Long</PARAMETER>
  <PARAMETER NAME="preserveOriginalContent"
    COMMENT="Should the document preserve the original content"
    DEFAULT="false">java.lang.Boolean</PARAMETER>
  <PARAMETER NAME="collectRepositioningInfo"
    COMMENT="Should the document collect repositioning information"
    DEFAULT="false">java.lang.Boolean</PARAMETER>
  <ICON>lr.gif</ICON>
</RESOURCE>


<RESOURCE>
  <NAME>Document Reset PR</NAME>
  <CLASS>gate.creole.annotdelete.AnnotationDeletePR</CLASS>
  <COMMENT>Document cleaner</COMMENT>
  <PARAMETER NAME="document" RUNTIME="true">gate.Document</PARAMETER>
  <PARAMETER NAME="annotationTypes" RUNTIME="true"
    OPTIONAL="true">java.util.ArrayList</PARAMETER>
</RESOURCE>
```

Parameters may be optional, and may have default values (and may have comments to describe their purpose, which is displayed by the GUI during interactive parameter setting).

Some PR parameters are execution time (`RUNTIME`), some are initialisation time. E.g. at execution time a doc is supplied to a language analyser; at initilisation time a grammar may be supplied to a language analyser.

Each parameter has a type, which may be the type of another resource, or a Java built in. Attributes of parameters:

**NAME:** name of the property that the parameter refers to; if supplied it will change the name that the initialisation routines assume are available to get/set on the resource (which are normally based on the value of the parameter, i.e. on the type of the parameter). The name must be identical to the property of the resource that the parameter relates to.

**DEFAULT:** default value.

**RUNTIME:** doesn't need setting at initialisation time, but must be set before calling `execute()`. Only meaningfull for PRs

**OPTIONAL:** not required

**COMMENT:** for display purposes

Visual Resources also have a `GUI` tag, which describes the resource (PR or LR) that it displays, whether it is the main viewer for that resource or not (main viewers are the first tab displayed for a resource) and whether the VR should go in the small viewers window or the large one. For example:

```
<RESOURCE>
  <NAME>Features Editor</NAME>
  <CLASS>gate.gui.FeaturesEditor</CLASS>
  <!-- type values can be "large" or "small"-->
  <GUI TYPE="large">
    <MAIN_VIEWER/>
    <RESOURCE_DISPLAYED>gate.util.FeatureBearer</RESOURCE_DISPLAYED>
  </GUI>
</RESOURCE>
```

More information:

- To collect PRs into an application and run them, see section 2.11.
- GATE's internal creole.xml file[13] (note that there are no JAR entries there, as the file is bundled with GATE itself).

## 2.11 [D] Create and Run an Application

Once all the resources have been loaded, an application can be created and run. Right click on "Applications" and select "New" and then either "Corpus Piepline" or "Pipeline". A pipeline application can only be run over a single document, while a corpus pipeline can be run over a whole corpus.

To build the pipeline, double click on it, and select the resources needed to run the application (you may not necessarily wish to use all those which have been loaded). Transfer the necessary components from the set of "loaded components" displayed on the left hand side of the main window to the set of "selected components" on the right, by selecting each component and clicking on the left and right arrows, or by double-clicking on each component. Ensure that the components selected are listed in the correct order for processing (starting from the top). If not, select a component and move it up or down the list using the up/down

[13] `http://gate.ac.uk/gate/src/gate/resources/creole/creole.xml`

arrows at the bottom of the pane. Ensure that any parameters necessary are set for each processing resource (by clicking on the resource from the list of selected resources and checking the relevant paramters from the pane below). For example, if you wish to use annotation sets other than the Default one, these must be defined for each processing resource. Note that if a corpus pipeline is used, the corpus needs only to be set once, using the drop-down menu beside the "corpus" box. If a pipeline is used, the document must be selected for each processing resource used. Finally, right-click on "Run" to run the application on the document or corpus.

## 2.12 [D] View Annotations

To view a document, double click on the filename in the left hand pane. Note that it may take a few seconds for the text to be displayed if it is long. The annotation types belonging to each annotation set are displayed to the right of the text. If no application has been run, the only annotations to be displayed will be those corresponding to the document format analysis performed automatically by Gate on loading the document (e.g. HTML or XML tags). If an application has been run, other annotation types and/or annotation sets may also be present. The fonts and colours of the annotations can be edited by double clicking on the annoation name

Select the annotation types to be viewed by clicking on the appropriate checkbox(es). The text segments corresponding to these annotations will be highlighted in the main text window.

Annotations relating to coreference (if relevant) are displayed separately. If the Orthomatcher has been run, a Coreference box will appear above the main window. Clicking on this will bring up a pane containing coreference annotations, below the pane containing the other annotations. These annotations can be viewed in exactly the same way.

Descriptions of the annotations are simultaneously displayed in the bottom pane. These lists can be sorted in ascending and descending order by any column, by clicking on the corresponding column heading. An arrow will appear indicating the direction of the sorting. Clicking on an entry in the table will also highlight the respective matching text portion.

Right clicking on some part of the text in the main window will bring up a box containing a list of the annotations associated with it. Selecting one of these annotation types will highlight the relevant annotation description in the lower pane, if present. If not present (because the corresponding annotation on the right hand pane has not been selected), this annotation on the right will then be automatically selected and all relevant text in the main window will be appropriately highlighted.

Although there is no cursor displayed in the various windows, they can all be scrolled using the keyboard arrows, as well as by using the scrollbars.

At any time, the main viewer can also be used to display other information, such as Messages, by clicking on the header at the top of the main window. If an error occurs in processing, the messages tab will flash red, and an additional popup error message may also occur.

## 2.13 [D] Do Information Extraction with ANNIE

This section describes how to load and run ANNIE (see Chapter 6) from the development environment. To embed ANNIE in other software, see section 2.20.

From the File menu, select "Load ANNIE system". To run it in its default state, choose "With Defaults". This will automatically load all the ANNIE resources, and create a corpus pipeline called ANNIE with the correct resources selected in the right order, and the default input and output annotation sets.

If "Without Defaults" is selected, the same processing resources will be loaded, but a popup window will appear for each resource, which enables the user to specify a name and location for the resource. This is exactly the same procedure as for loading a processing resource individually, the difference being that the system automatically selects those resources contained within ANNIE. When the resources have been loaded, a corpus pipeline called ANNIE will be created as before.

The next step is to add a corpus (see Section 2.9.1), and select this corpus from the drop-down Corpus menu in the Serial Application editor. Finally click on Run (from the Serial Application editor, or by right clicking on the application name and selecting "Run"). To view the results, double click on the filename in the left hand pane.

## 2.14 [D] Create and Edit Test Data

Since many NLP algorithms require annotated corpora for training, GATE's development environment provides easy-to-use and extendable facilities for text annotation. The annotation can be done manually by the user or semi-automatically by running some processing resources over the corpus and then correcting/adding new annotations manually. Depending on the information that needs to be annotated, some ANNIE modules can be used or adapted to bootstrap the corpus annotation task.

Since manual annotation is a difficult and error-prone task, GATE tries to make it simple to use and yet keep it flexible. To add a new annotation, select the text with the mouse (e.g. "Mr. Clever") and then click on the desired annotation type (e.g. Person), which is shown in the list of types on the right hand side of the document viewer. If however the desired annotation type does not already appear there or the user wants to associate more detailed information with the annotation (not just its type), then an annotation editing dialogue can

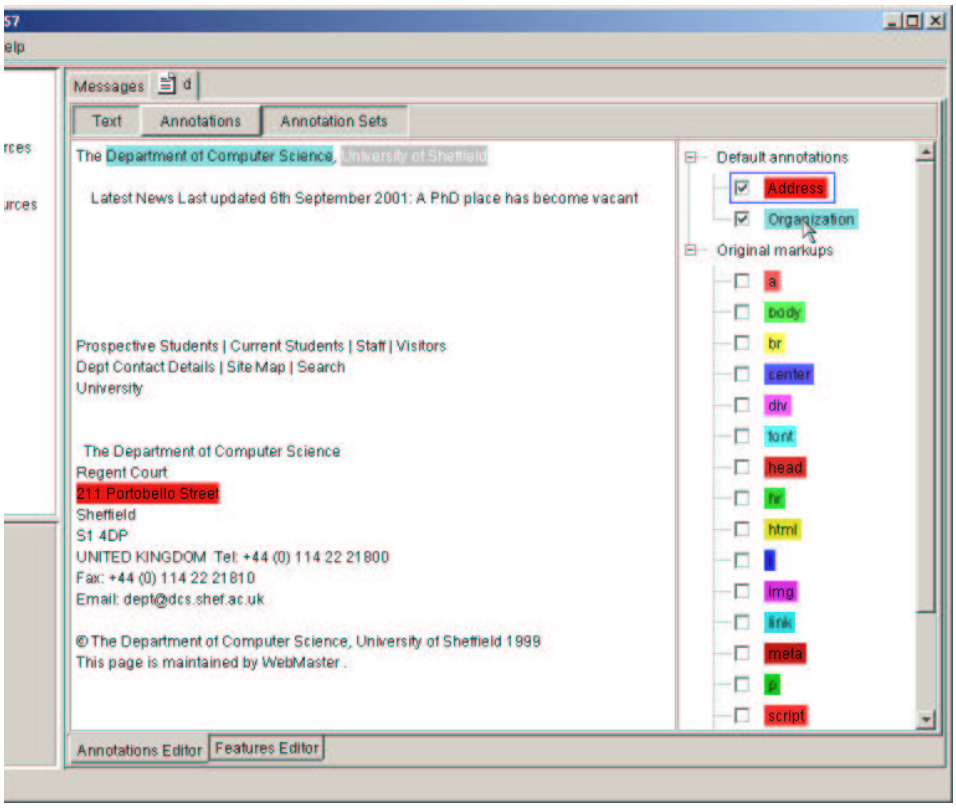



Figure 2.4: Adding an Organization annotation to the Default Annotation Set

be used.

Figure 2.4 demonstrates adding the Organization annotation for the string "University of Sheffield" (highlighted in grey) to the Default Annotation set.

### 2.14.1 Schema Annotation Editor

To use the Schema Annotation Editor, select the text to be annotated, right click and select the annotation set to which the annotation should be added. An "Edit Annotation" popup window should appear, offering the choice between a Schema Annotation Editor (the default) and an Unrestricted Annotation Editor.

If an annotation schema already exists for an annotation type, the Schema Annotation Editor can be used to add a new annotation to a document. For information about annotation schemas, see Section 4.4.1. To see whether an annotation schema has been loaded, it will be present in the dropdown list of annotation types in the Schema Annotation Editor. If the relevant annotation schema does not exist, it must either be created and/or loaded (see

Section 2.22), or the Unrestricted Annotation Editor can be used (see Section 2.14.2.

To add an annotation, select the annotation type from the dropdown list. If the annotation type can have features associated with it (according to its definition in the annotation schema), a list of possible features will appear. Use the arrows to select those features required, and fill in the values as appropriate. Click OK to create the annotation and return to the main Gate window. The new annotation should now appear in the annotations list.

The Annotation Schema Editor can also be used to edit an existing annotation , e.g. to change the values of the features associated with it. This is done by right clicking on the annotation in the text, selecting the annotation set, annotation name, and "Select", and then right clicking on the annotation in the lower pane and selecting "Edit". This will bring up the relevant Annotation Schema Editor for that annotation.

### 2.14.2 Unrestricted Annotation Editor

If an annotation schema does not exist, or if the user wishes to have more flexibility in defining the annotation (e.g. to add new features not present in the schema), the Unrestricted Annotation Editor should be used. From the "Edit Annotations" window, select "Unrestricted annoation editor", and enter the annotation name and feature names and values. It is important to remember to click on "Add feature" before clicking on OK and returning to the main Gate window, if any features have been added, otherwise these will be lost. To add more than one feature, click on "Add Feature" after each feature has been added.

### 2.14.3 Saving the test data

The data can either be dumped out as a file (see Section 2.23 or saved in a data store (see Section 2.15.

## 2.15 [D] Save and Restore LRs in Data Stores

To save a text in a data store, a new data store must first be created if one does not already exist. Create a data store by right clicking on Data Store in the left hand pane, and select the option "Create Data Store". Select the data store type you wish to use. Create a directory to be used as the data store (note that the data store is a directory and not a file). Save the text to the data store by right clicking on the document name and selecting the "Save to..." option (giving the name of the datastore created earlier).

To load a document from a data store, do not try to load it as a language resource. Instead, open the data store by right clicking on Data Store in the left hand pane, select "Open Data

Store" and choose the data store to open. The filenames contained in the data store should appear in the left hand pane. Double click on a file to open it. Once opened, the file will then appear under Language Resources in the left hand pane. Double click on this file to view it in the main window. It can be treated in the same way as any other document.

## 2.16 [D] Save Resource Parameter State to File

Resources, and applications that are made up of them, are created based on the settings of their parameters (see section 2.9). It is possible to save the data used to create a an application to file and re-load it later. To save the application to file, right click on it in the resources tree and select "Save application state", which will give you a file creation dialogue.

To restore the application later, select "Restore application from file" from the "File" menu.

Note that the data that is saved represents how to *recreate* an application – not the resources that make up the application itself. So, for example, if your application has a resource that initialises itself from some file (e.g. a grammar) then that file must still exist when you restore the application.

## 2.17 [D,F] Perform Evaluation with the Annotation-Diff tool

Section 8 describes the theory behind this tool.

### 2.17.1 GUI

The annotation tool is activated by selecting it from the Tools menu at the top of the window. It will appear in a new window. Select the key and response documents to be used (note that both must have been previously loaded into the system), the annotation sets to be used for each, and the annotation type to be evaluated.

Note that the tool automatically intersects all the annotation types from the selected key annotation set with all types from the response set.

On a separate note, you can perform a diff on the same document, between two different annotation sets. One annotation set could contain the key type and another could contain the response one.

After the type has been selected, the user is required to decide how the features will be

compared. It is important to know that the tool compares them by analyzing if features from the key set are contained in the response set. It checks for both the feature name and feature value to be the same.

There are three basic options to select:

- To take all the features from the key set into consideration
- To take only the user selected ones
- To ignore all the features from the key set.

If false positives are to be measured, select the annotation type (and relevant annotation set) to be used as the denominator (normally, Token or Sentence). The weight for the F-Measure can also be changed - by default it is set to 0.5 (i.e. to give precision and recall equal weight). Finally, click on "Evaluate" to display the results. Note that the window may need to be resized manually, by dragging the window edges or internal bars as appropriate).

In the main window, the key and response annotations will be displayed. They can be sorted by any category by clicking on the relevant column header. The key and response annotations will be aligned if their indices are identical, and are color coded according to the legend displayed.

Precision, recall, F-measure and false positives are also displayed below the annotation tables, each according to 3 criteria - strict, lenient and average. See sections 8.1 and 8.4 for more details about the evaluation metrics.

### 2.17.2 API

Since Annotation Diff is a tool, not a processing resource, it needs to be constructed directly, via its constructor. Then all its parameters are set using the respective mutator methods (i.e., setXXX()), and finally the init() method needs to be called to make it calculate the statistics.

*Example:*

```
AnnotationDiff annotDiff = new AnnotationDiff();
annotDiff.setKeyDocument(keyDocument);
annotDiff.setResponseDocument(responseDocument);
annotDiff.setAnnotationSchema(annotationSchema);
annotDiff.setKeyAnnotationSetName(
              GateConstants.ORIGINAL_MARKUPS_ANNOT_SET_NAME);
annotDiff.setResponseAnnotationSetName(
```

```
                GateConstants.ORIGINAL_MARKUPS_ANNOT_SET_NAME);
annotDiff.init();
```

It is important to know that its `init()` method performs the diff between two sets of annotations. So, after the initialization, one could use the getZZZ() methods to read the results of the evaluation.

### 2.17.3 Annotation Diff parameters

All Annotation Diff parameters by default are initialized to **null** (except textMode) and they are as follows:

keyDocument = the document holding the key annotation sets that will be used in the evaluation. It is a gate.Document and if it is null then init() will throw a ResourceInstantiationException.

responseDocument = the document processed by GATE, containing annotation sets generated by the system. AnnotationDiff will behave the same as the keyDocument if this parameter it is set to null.

annotationSchema = It represent an gate.creole.AnnotationSchema object that describes an annotation. The information used by the Annotation Diff is the type of the annotations being evaluated.

keyAnnotationSetName = a String representing the name of the annotation set from the key document that also holds annotations of the type specified in the annotationSchema parameter. If it's set to null then the default annotation set will be used.

responseAnnotationSetName = same as for the key annotation set.

keyFeatureNamesSet = a Set containing keys (the keys of the features from the annotation key set taken into consideration when performing the diff). If it is null, then all the features from the key set will be taken into consideration. If it is an empty set, then no feature will be taken into consideration. Otherwise only the keys specified in this set will be taken into consideration by the evaluation process.

textMode = if set to **true**, it will not construct the graphic components (like for example the table displaying the results). The structure used for building the graphic representation will be constructed and AnnotationDiff will provide means to access its results through its get...() methods. By default, this parameter is set to **false**.

### 2.17.4 Reading the results from the Annotation Diff

By using the API all the calculated measures cam be accessed through getZZZ() methods.

*Example:*

```
annotDiff.getPrecisionAverage();
annotDiff.getPrecisionLenient();
annotDiff.getPrecisionStrict();
...
```

All four types of classified annotations can be accessed using only one special get method called getAnnotationsOfType.

*Example:*

```
AnnotationSet correctAnnot=
annotDiff.getAnnotationsOfType(AnnotationDiff.CORRECT_TYPE);

AnnotationSet partiallyCorrectAnnot=
annotDiff.getAnnotationsOfType(AnnotationDiff.PARTIALLY_CORRECT_TYPE);

AnnotationSet missingAnnot=
annotDiff.getAnnotationsOfType(AnnotationDiff.MISSING_TYPE);

AnnotationSet spuriousAnnot=
annotDiff.getAnnotationsOfType(AnnotationDiff.SPURIOUS_TYPE);
```

## 2.18 [D] Use the Corpus Benchmark Evaluation tool

The Corpus Benchmark tool can be run in two ways: standalone and GUI mode. Section 8.3 describes the theory behind this tool.

### 2.18.1 GUI mode

To use the tool in GUI mode, select "Corpus Benchmark Tool" from the Options menu. Select "Default mode" to compare the stored processed set with the current processed set and the human-annotated set. Select "human marked against stored processing results" to compare the stored processed set with the human-annotated set. Select "human marked against current processing results" to compare the current processed set with the human-annotated set.

Once the mode has been selected, choose the directory where the corpus is to be found. The corpus must have a directory structure consisting of "clean" and "marked" subdirectories). The clean directory should contain the raw texts; the marked directory shuold contain the human-annotated texts.

If the tool is to be used in Default or Current mode, the corpus must first be processed with the current set of resources. This is done by selecting "Store corpus for future evaluation" from the Corpus Benchmark Tools. Select the corpus to be processed (from the top of the subdirectory structure, i.e. the directory containing the marked and stored subdirectories). If a "processed" subdirectory exists, the results will be placed there; if not, one will be created.

Once the corpus has been processed, the tool can be run in Default or Current mode. The resulting HTML file will be output in the main GATE messages window. This can then be pasted into a text editor and viewed in an internet browser for easier viewing.

The tool can be used either in verbose or non-verbose mode, by selecting the verbose option from the menu. In verbose mode, any score below the user's pre-defined threshold (stored in corpus_tool.properties file) will show the relevant annotations for that entity type, thereby enabling the user to see where problems are occurring. See section 2.18.3 for a description of the corpus_tool.properties file.

### 2.18.2 Standalone mode

Alternatively, the tool can be run in standalone mode, using the following commands:

- To process the corpus, issue the command

  ```
  gate -e -generate corpusname
  ```

  (where 'corpusname' is the name of the corpus)

- To run in Stored mode, issue the command

  ```
  gate -e [-verbose] -marked\_stored corpusname
  ```

- To run in Current mode, issue the command

  ```
  gate -e [-verbose] -marked\_clean corpusname
  ```

- To run in Default mode, issue the command

  ```
  gate -e [-verbose] corpusname
  ```

The tool can be run in verbose mode for any of these options using the [-verbose] flag. The results can be piped to an html file and viewed with an internet browser.

### 2.18.3 How to define the properties of the benchmark tool

The properties of the benchmark tool are defined in the file corpus_tool.properties which should be located in the firectory from which Gate is run. The file specifies the location for each processing resource to be used in the application. The pathname should have the same format as that used when specifying the location of a resource n the GUI, e.g.

```
tokeniserRulesURL=gate:/creole/tokeniser/DefaultTokeniser.rules
```

or

```
tokeniserRulesURL=file:/user/gate/src/gate/resources/creole/tokeniser/DefaultTokeniser.
```

Other parameters which can be set here are:

- whether the gazetteer is case sensitive or not
- the name of the textTag for the Annotation Set Transfer (if used)
- the threshold for the verbose mode (by default this is set to 0.5)
- the name of the annotation set containing the human-marked annotations

## 2.19 [D] Write JAPE Grammars

JAPE is a language for writing regular expressions over annotations, and for using patterns matched in this way as the basis for creating more annotations. JAPE rules compile into finite state machines. GATE's built-in Information Extraction tools use JAPE (amongst other things). For information on JAPE see:

- chapter 5 describes how to write JAPE rules;
- chapter 6 describes the built-in IE components;
- appendix B describes how JAPE is implemented and formally defines the language's grammar;
- appendix C describes the default Named Entity rules distributed with GATE.

## 2.20 [F] Embed NLE in other Applications

Embedding GATE-based language processing in other applications is straightforward:

- add `gate.jar` to the `CLASSPATH`, e.g. `CLASSPATH=/home/hamish/gate/bin/gate.jar`;
- tell Java that the GATE Unicode Kit is an extension (`-Djava.ext.dirs=/home/hamish/gate/bin/e` for example);
- initialise GATE with `gate.Gate.init()`;
- program to the framework API.

For example, this code will create the ANNIE extraction system:

```
public static void main(String args[]) throws GateException, IOException {
  // initialise the GATE library
  Gate.init();

  // initialise ANNIE
  // create a corpus pipeline controller to run ANNIE with
  annieController =
    (SerialAnalyserController) Factory.createResource(
    "gate.creole.SerialAnalyserController", Factory.newFeatureMap(),
    Factory.newFeatureMap(), "ANNIE_" + Gate.genSym()
  );

  // load each PR as defined in ANNIEConstants
  for(int i = 0; i < ANNIEConstants.PR_NAMES.length; i++) {
    FeatureMap params = Factory.newFeatureMap(); // use default parameters
    ProcessingResource pr = (ProcessingResource)
    Factory.createResource(ANNIEConstants.PR_NAMES[i], params);

    // add the PR to the pipeline controller
    annieController.add(pr);
  } // for each ANNIE PR

....
```

A longer example of embedding ANNIE is available at `http://gate.ac.uk/GateExamples/doc/`.

## 2.21 [D,F] Add support for a new document format

In order to add a new document format, one needs to extend the `gate.DocumentFormat` class and to implement an abstract method called:

```
public void unpackMarkup(Document doc) throws
DocumentFormatException
```

This method is supposed to implement the functionality of each format reader and to create annotation on the document. Finally the document's old content will be replaced with a new one containing only the text between markups (see the GATE API documentation for more details on this method functionality).

If one needs to add a new textual reader will extend the gate.corpora. TextualDocument-Format and override the `unpackMarkup(doc)` method.

This class needs to be implemented under the Java bean specifications because it will be instantiated by GATE using `Factory.createResource()` method.

The `init()` method that one needs to add and implement is very important because in here the reader defines its means to be selected successfully by GATE. What one need to do is to add some specific information into certain static maps defined in `DocumentFormat` class, that will be used at reader detection time.

After that, a definition of the reader will be placed into the one's creole.xml file and the reader will be available to GATE.

We present for the rest of the section a complete three steps example of adding such a reader. The reader we describe in here is an XML reader.

**Step 1**

Create a new class called `XmlDocumentFormat` that extends `gate.corpora.TextualDocumentFormat`.

**Step 2**

Implement the `unpackMarkup(Document doc)` which performs the required functionality for the reader. Add XML detection means in init() method:

```
public Resource init() throws ResourceInstantiationException{
  // Register XML mime type
  MimeType mime = new MimeType("text","xml");
  // Register the class handler for this mime type
  mimeString2ClassHandlerMap.put(mime.getType()+ "/" + mime.getSubtype(),
                                                                    this);
  // Register the mime type with mine string
```

```
  mimeString2mimeTypeMap.put(mime.getType() + "/" + mime.getSubtype(), mime);
  // Register file sufixes for this mime type
  suffixes2mimeTypeMap.put("xml",mime);
  suffixes2mimeTypeMap.put("xhtm",mime);
  suffixes2mimeTypeMap.put("xhtml",mime);
  // Register magic numbers for this mime type
  magic2mimeTypeMap.put("<?xml",mime);
  // Set the mimeType for this language resource
  setMimeType(mime);
  return this;
}// init()
```

More details about the information from those maps can be found in Section 4.5.1

**Step 3**

Add the following creole definition in the creole.xml document.

```
<RESOURCE>
  <NAME>My XML Document Format</NAME>
  <CLASS>mypackage.XmlDocumentFormat</CLASS>
  <AUTOINSTANCE/>
  <PRIVATE/>
</RESOURCE>
```

More information on the operation of GATE's document format analysers may be found in section 4.5.

## 2.22 [D,F] Create a New Annotation Schema

**GUI**

An annotation schema file can be loaded or unloaded in GATE just like any other language resource. Once loaded into the system, the SchemaAnnotationEditor will use this definition when creating or editing annotations.

**API**

Another way to bring an annotation schema inside GATE is through creole.xml file. By using the AUTOINSTANCE element, one can create instances of resources defined in creole.xml. The gate.creole.AnnotationSchema (which is the Java representation of an annotation schema file) initializes with some predefined annotation definitions (annotation schemas) as specified by the GATE team.

*Example from GATE's creole.xml:*

```
<!-- Annotation schema -->
<RESOURCE>
 <NAME>Annotation schema</NAME>
  <CLASS>gate.creole.AnnotationSchema</CLASS>
  <COMMENT>An annotation type and its features</COMMENT>
  <PARAMETER NAME="xmlFileUrl" COMMENT="The url to the definition file"
   SUFFIXES="xml;xsd">java.net.URL</PARAMETER>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/DateSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/FacilitySchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/TokenSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/SyntaxTreeNodeSchema.xml"/></AUTOINSTANCE>
  <AUTOINSTANCE><PARAM NAME ="xmlFileUrl"
   VALUE ="gate:/creole/schema/CorefSchema.xml"/></AUTOINSTANCE>
</RESOURCE>
```

In order to create a gate.creole.AnnotationSchema object from a schema annotation file, one must use the gate.Factory class.

Eg:

```
FeatureMap params = new FeatureMap();
param.put("xmlFileUrl",annotSchemaFile.toURL());
AnnotationSchema annotSchema =
Factory.createResurce("gate.creole.AnnotationSchema", params);
```

**Note:** All the elements and their values must be written in lower case, as XML is defined as case sensitive and the parser used for XML Schema inside GATE searches is case sensitive.

In order to be able to write XML Schema definitions, the ones defined in GATE (resorces/creole/schema) can be used as a model, or the user can have a look at *http://www.w3.org/2000/10/XMLSchema* for a proper description of the semantics of the elements used.

Some examples of annotation schemas are given in Section 4.4.1.

## 2.23 [D] Dump Results to File

There are three main ways to dump out the results of, for example, some language analysis or Information Extraction process running over documents:

1. preserving the original document format, with optional added annotations;

2. in GATE's own XML serialisation format (including all the annotations on the document);

3. by writing your own dump algorithm as a ProcessingResource.

This section describes how to use the first two options.

Both types of data export are available in the popup menu triggered by right-clicking on a document in the resources tree (see section 2.4): type 1 is called 'save preserving format' and type 2 is called 'save as XML'.

Selecting the save as XML option leads to a file open dialogue; give the name of the file you want to create, and the whole document and all its data will be exported to that file. If you later create a document from that file, the state will be restored. (**Note:** because GATE's annotation model is richer than that of XML, and because our XML dump implementation sometimes cuts corners[14], the state may not be identical after restoration. If your intention is to store the state for later use, use a DataStore instead.)

The save preserving format option also leads to a file dialogue; give a name and the data you require will be dumped into the file. The difference is that the file will preserve the original format of the source document. You can add annotations to the dump file: if there is a document viewer open in the main resource viewer area (see section 2.4), then any annotations that are selected (i.e. are visible in the table at the bottom of the viewer) will be included in the output dump. This is the best way to use the system to add markup based on some analysis process: select those annotations in the document viewer, save preserving format and you will have a file identical to the original source document with just the annotations you selected added. By default, the added annotations will contain no feature data; if you want the process to also dump features, set the 'Include annotation features...' option in the advanced options dialogue (see section 2.5). Note that GATE's model of annotation allows graph structures, which are difficult to represent in XML (XML is a tree-structured representation format). During the dump process, annotations that cross each other in ways that can't be represented straightforwardly in XML will be discarded, and a warning message printed.

## 2.24 [D] Stop GUI 'Freezing' on Linux

There is a problem with some versions of Linux that causes the GUI to appear to freeze. The problem occurs when you take some action, like loading a resource or browsing for a file, that pops up a dialogue box. This box sometimes fails to appear in a visible area of the screen, at which point the rest of the GUI waits for you to do something intelligent with the

[14]Gorey details: features of annotations and documents in GATE may be any virtually any Java object; serialising arbitrary binary data to XML is not simple; instead we serialise them as strings, and therefore they will be re-loaded as strings.

dialogue box, while you wait for the GUI to do something. This is an excellent feature for those without tight deadlines to meet, and the best solution is to stop work and go home for a long while. Alternatively, you can play 'hunt the dialogue box'.

This feature is available totally free of charge.

## 2.25 [D] Stop GATE Restoring GUI Sessions/Options

GATE will remember GUI options and the state of the resource tree when it exits. The options are saved by default; the session state is not saved by default. This default behaviour can be changed from the "Advanced" tab of the "Configuration" choice on the "Options" menu.

If a problem occurs and the saved data prevents GATE from starting, you can fix it by deleting the configuration and session data files. These are stored in your home directory, and are called `gate.xml` and `gate.sesssion` or `.gate.xml` and `.gate.sesssion` depending on platform. On Windoze your home is:

**95, 98, NT:** Windows Directory/profiles/username

**2000, XP:** Windows Drive/Documents and Settings/username

## 2.26 Work with Unicode

GATE provides various facilities for working with Unicode beyond those that come as default with Java[15]:

1. a Unicode editor with input methods for many languages;
2. use of the input methods in all places where text is edited in the GUI;
3. a development kit for implementing input methods;
4. ability to read diverse character encodings.

**1 using the editor:**
In the GUI, select 'Unicode editor' from the 'Tools' menu. This will display an editor window,

[15]Implemented by Valentin Tablan, Mark Leisher and Markus Kramer. Initial version developed by Mark Leisher.

and, when a language with a custom input method is selected for input (see next section), a virtual keyboard window with the characters of the language assigned to the keys on the keyboard. You can enter data either by typing as normal, or with mouse clicks on the virtual keyboard.

**2 configuring input methods:**
In the editor and in GATE's main window, the 'Options' menu has an 'Input methods' choice. All supported input languages (a superset of the JDK languages) are available here. Note that you need to use a font capable of displaying the language you select. By default GATE will choose a Unicode font if it can find one on the platform you're running on. Otherwise, select a font manually from the 'Options' menu 'Configuration' choice.

**3 using the development kit:**
GUK, the GATE Unicode Kit, is documented at `http://gate.ac.uk/gate/doc/javadoc/guk/package-`

**4 reading different character encodings:**
When you create a document from a URL pointing to textual data in GATE, you have to tell the system what character encoding the text is stored in. By default, GATE will set this parameter to be the empty string. This tells Java to use the default encoding for whatever platform it is running on at the time – e.g. on Western versions of Windoze this will be ISO-8859-1, and Eastern ones ISO-8859-9. A popular way to store Unicode documents is in UTF-8, which is a superset of ASCII (but can still store all Unicode data); if you get an error message about document I/O during reading, try setting the encoding to UTF-8, or some other locally popular encoding. (To see a list of available encodings, try opening a document in GATE's unicode editor – you will be prompted to select an encoding.)

## 2.27 Work with Oracle

GATE's Oracle layer is documented separately in `http://gate.ac.uk/gate/doc/persistence.pdf`. Note that running an Oracle installation is not for the faint-hearted!

# Chapter 3

# CREOLE: the GATE Component Model

> ... Noam Chomsky's answer in *Secrets, Lies and Democracy* (David Barsamian 1994; Odonian) to "What do you think about the Internet?"
>
> "I think that there are good things about it, but there are also aspects of it that concern and worry me. This is an intuitive response – I can't prove it – but my feeling is that, since people aren't Martians or robots, direct face-to-face contact is an extremely important part of human life. It helps develop self-understanding and the growth of a healthy personality.
>
> "You just have a different relationship to somebody when you're looking at them than you do when you're punching away at a keyboard and some symbols come back. I suspect that extending that form of abstract and remote relationship, instead of direct, personal contact, is going to have unpleasant effects on what people are like. It will diminish their humanity, I think."
>
> Chomsky, quoted at `http://photo.net/wtr/dead-trees/53015.htm`.

The GATE architecture is based on components: reusable chunks of software with well-defined interfaces that may be deployed in a variety of contexts. The design of GATE is based on an analysis of previous work on infrastructure for LE, and of the typical types of software entities found in the fields of NLP and CL (see in particular chapters 4–6 of [Cunningham 00]). Our research suggested that a profitable way to support LE software development was an architecture that breaks down such programs into components of various types. Because LE practice varies very widely (it is, after all, predominantly a research field), the architecture must avoid restricting the sorts of components that developers can plug into the infrastructure. The GATE framework accomplishes this via an adapted version of the *Java Beans* component framework from Sun. Section 3.2 describes Java's component model, Java Beans; section 3.3 describes GATE's extended Beans model.

GATE components may be implemented by a variety of programming languages and databases, but in each case they are represented to the system as a Java class. This class may do nothing other than call the underlying program, or provide an access layer to a database; on the other hand it may implement the whole component.

GATE components are one of three types:

- LanguageResources (LRs) represent entities such as lexicons, corpora or ontologies;
- ProcessingResources (PRs) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers;
- VisualResources (VRs) represent visualisation and editing components that participate in GUIs.

Section 3.4 discusses the disctinction between Language Resources and Processing Resources. Collectively, the set of resources integrated with GATE is known as **CREOLE**: a Collection of REusable Objects for Language Engineering.

In the rest of this chapter:

- section 3.5 describes the lifecycle of GATE components;
- section 3.6 describes how Processing Resources can be grouped into applications;
- section 3.7 describes the relationship between Language Resources and their data stores;
- section 3.8 summarises GATE's set of built-in components.

## 3.1 The Web and CREOLE

GATE allows resource implementations and Language Resource persistent data to be distributed over the Web, and uses XML for configuration of resources (and GATE itself).

Resource implementations are stored at a URL (when the resources are in the local file system this can be a `file:/` URL). When the URL is given to GATE the `creole.xml` component configuration file is sucked down the pipe and the resource information added to the CREOLE register. When a user requests an instantiation of a resource, the class files are sucked up too, and an object created in the local virtual machine.

Language resource data can be stored in binary serialised form in the local file system, or in an RDBMS like Oracle. In the latter case, communication with the database is over JDBC[1],

[1]The Java DataBase Connectivity layer.

allowing the data to be located anywhere on the network (or anywhere you can get Oracle running, that is!).

## 3.2 Java Beans: a Simple Component Architecture

All GATE resources are *Java Beans*, the Java platform's model of software components. Beans are simply Java classes that obey certain interface conventions. These conventions allow development tools such as GATE, or Borland JBuilder, to manipulate software components without knowing very much about them. The advantage of this is that users of such systems can extend them in diverse ways without having to touch the underlying core of the development tools.

The key parts of the Java Beans specification as used in GATE are:

- accessor and mutator methods for data members are named after those members plus `get` and `set` (meaning that the tool can figure out how to use a member, or *property*, of a bean, from information provided by Java reflection);
- beans must have no-argument constructors (so that tools can construct instances of beans without knowing about complex initialisation parameters).

The rest of this section says a little more about the Beans specification; skip to the next if you're only interested in how it works in GATE.

Quoting from [Campione *et al.* 98] at Sun's Java website[2]:

> The JavaBeans API makes it possible to write component software in the Java programming language. Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools. JavaBean components are known as *Beans*.

In this context we may think of the GATE development environment as a 'builder tool'. While the emphasis in the quoted text is on visual representation of components, note that GATE (and other) beans can also be plugged together 'invisibly'; this is what the framework does and how GATE beans are typically deployed into other applications.

> Components expose their features (for example, public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific *design patterns*. A JavaBeans-enabled builder

[2] `http://java.sun.com/docs/books/tutorial/javabeans/whatis/beanDefinition.html`

> tool can then examine the Bean's patterns, discern its features, and expose those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a Bean from the toolbox, drop it into a form, modify it's appearance and behavior, define its interaction with other Beans, and compose it and other Beans into an applet, application, or new Bean. All this can be done without writing a line of code.

In GATE you develop sets of beans that do language processing tasks and then the framework wires them together without any code from you.

> - Builder tools discover a Bean's features (that is, its properties, methods, and events) by a process known as *introspection*. Beans support introspection in two ways:
>   - By adhering to specific rules, known as *design patterns*, when naming Bean features. The `Introspector` class examines Beans for these design patterns to discover Bean features. The `Introspector` class relies on the core reflection API. ...

The next section describes GATE's extended beans model.

## 3.3 The GATE Framework

We can think of the GATE framework as a backplane into which plug beans-based CREOLE components. The user gives the system a list of URLs to search when it starts up, and components at those locations are loaded by the system. (To be precise only their configuration data is loaded to begin with; the actual classes are loaded when the user requests the instantiation of a resource.)

The backplane performs these functions:

- component discovery, bootstrapping, loading and reloading;
- management and visualisation of native data structures for common information types;
- generalised data storage and process execution.

A set of components plus the framework is a deployment unit which can be embedded in another application.

The key task of the development environment is to facilitate constructing components, and viewing and measuring their results.

## 3.4 Language Resources and Processing Resources

This section describes in more detail the *Language Resource* and *Processing Resource* terminology introduced earlier. If you're happy with these terms you can safely skip this section.

Like other software, LE programs consist of data and algorithms. The current orthodoxy in software development is to model both data and algorithms together, as *objects*[3]. Systems that adopt the new approach are referred to as Object-Oriented (OO), and there are good reasons to believe that OO software is easier to build and maintain than other varieties [Booch 94, Yourdon 96].

In the domain of human language processing R&D, however, the terminology is a little more complex. Language data, in various forms, is of such significance in the field that it is frequently worked on independently of the algorithms that process it. For example: a treebank[4] can be developed independently of the parsers that may later be trained from it; a thesaurus can be developed independently of the query expansion or sense tagging mechanisms that may later come to use it. This type of data has come to have its own term, *Language Resources* (LRs) [LREC-1 98], covering many data sources, from lexicons to corpora.

In recognition of this distinction, we will adopt the following terminology:

**Language Resource (LR):** refers to data-only resources such as lexicons, corpora, thesauri or ontologies. Some LRs come with software (e.g. Wordnet has both a user query interface and C and Prolog APIs), but where this is only a means of accessing the underlying data we will still define such resources as LRs.

**Processing Resource (PR):** refers to resources whose character is principally programmatic or algorithmic, such as lemmatisers, generators, translators, parsers or speech recognisers. For example, a part-of-speech tagger is best characterised by reference to the process it performs on text. PRs typically *include* LRs, e.g. a tagger often has a lexicon; a word sense disambiguator uses a dictionary or thesaurus.

Additional terminology worthy of note in this context: *language data* refers to LRs which are at their core examples of language in practice, or 'performance data', e.g. corpora of texts or speech recordings (possibly including added descriptive information as markup); *data about language* refers to LRs which are purely descriptive, such as a grammar or lexicon.

PRs can be viewed as algorithms that map between different types of LR, and which typically use LRs in the mapping process. An MT engine, for example, maps a monolingual corpus into a multilingual aligned corpus using lexicons, grammars, etc.[5]

[3] Older development methods like Jackson Structured Design [Jackson 75] or Structured Analysis [Yourdon 89] kept them largely separate.

[4] A corpus of texts annotated with syntactic analyses.

[5] This point is due to Wim Peters.

Further support for the PR/LR terminology may be gleaned from the argument in favour of declarative data structures for grammars, knowledge bases, etc. This argument was current in the late 1980s and early 1990s [Gazdar & Mellish 89], partly as a response to what has been seen as the overly procedural nature of previous techniques such as augmented transition networks. Declarative structures represent a separation between data about language and the algorithms that use the data to perform language processing tasks; a similar separation to that used in GATE.

Adopting the PR/LR distinction is a matter of conforming to established domain practice and terminology. It does not imply that we cannot model the domain (or build software to support it) in an Object-Oriented manner; indeed the models in GATE are themselves Object-Oriented.

## 3.5 The Lifecycle of a CREOLE Resource

CREOLE resources exhibit a variety of forms depending on the perspective they are viewed from. Their implementation is as a Java class plus an XML metadata file living at the same URL. When using the development environment, resources can be loaded and viewed via the resources tree (left pane) and the "create resource" mechanism. When programming with the framework, they are Java objects that are obtained by making calls to GATE's `Factory` class. These various incarnations are the phases of a CREOLE resource's 'lifecycle'. Depending on what sort of task you are using GATE for, you may use resources in any or all of these phases. For example, you may only be interested in getting a graphical view of what GATE's ANNIE Information Extraction system (see chapter 6) does; in this case you will use the GUI to load the ANNIE resources, and load a document, and create an ANNIE application and run it on the document. If, on the other hand, you want to create your own resources, or modify the Java code of an existing resource (as opposed to just modifying its grammar, for example), you will need to deal with all the lifecylce phases.

The various phases may be summarised as:

**Creating a new resource from scratch (bootstrapping).** To create the binary image of a resource (a Java class in a JAR file), and the XML file that describes the resource to GATE, you need to create the appropriate `.java` file(s), compile them and package them as a `.jar`. The GATE development environment provides a bootstrap tool to start this process – see section 2.7. Alternatively you can simply copy code from an existing resource.

**Instantiating a resource in the framework.** To create a resource in your own Java code, use GATE's `Factory` class (this takes care of parameterising the resource, restoring it from a database where appropriate, etc. etc.). Section 2.8 describes how to do this.

**Loading a resource in the development environment.** To load a resource in the development environment, use the various "New ... resource" options from the `File` menu and elsewhere. See section 2.9.

**Resource configuration and implementation.** GATE's bootstrap tool will create an empty resource that does nothing. In order to achieve the behaviour you require, you'll need to change the configuration of the resource (by editing the `creole.xml` file) and/or change the Java code that implements the resource. See section 2.10.

More details of the specifics of tasks related to these phases are available in chapter 2.

## 3.6 Processing Resources and Applications

PRs can be combined into *applications*. Applications model a control strategy for the execution of PRs. In the framework applications are called 'controllers' accordingly.

Currently only sequential, or pipeline, execution is supported. There are two types of pipeline:

**Simple pipelines** simply group a set of PRs together in order and execute them in turn. The implementing class is called `SerialController`.

**Corpus pipelines** are specific for LanguageAnalysers – PRs that are applied to documents and corpora. A corpus pipeline opens each document in the corpus in turn, sets that document as a runtime parameter on each PR, runs all the PRs on the corpus, then closes the document. The implementing class is called `SerialAnalyserController`.

## 3.7 Language Resources and Datastores

Language Resources can be stored in Data Stores. Data Stores are an abstract model of disk-based persistence, which can be implemented by various types of storage mechanism. Currently two such mechanisms are implemented:

**Serial Data Stores** are based on Java's serialisation system, and store data directly into files and directories.

**Oracle Data Stores** store data into an Oracle RDBMS. For details of how to set up an Oracle DB for GATE, see `http://gate.ac.uk/gate/doc/persistence.pdf`.

Support for PostgreSQL is in development.

## 3.8 Built-in CREOLE Resources

GATE comes with various built-in components:

- Language Resources modelling Documents and Corpora, and various types of Annotation Schema – see chapter 4.
- Processing Resources that are part of the ANNIE system – see chapter 6.
- Visual Resources for viewing and editing corpora, annotations, etc.
- Other miscellaneous resources – see chapter 7.

Contributions to further developments gratefully received (unmarked low-denomination notes preferred). Bugs to `santa@northpole.org`.

# Chapter 4

# Corpora, Documents and Annotations

> Sometimes in life you've got to dance like nobody's watching.
> ...
> I think they should introduce 'sleeping' to the Olympics. It would be an excellent field event, in which the 'athletes' (for want of a better word) all lay down in beds, just beyond where the javelins land, and the first one to fall asleep and not wake up for three hours would win gold. I, for one, would be interested in seeing what kind of personality would be suited to sleeping in a competitive environment.
> ...
> Life is a mystery to be lived, not a problem to be solved.
>
> *Round Ireland with a Fridge*, Tony Hawks, 1998 (pp. 119, 147, 179).

This chapter documents GATE's model of corpora, documents and annotations on documents. Section 4.1 describes the simple attribute/value data model that corpora, documents and annotations all share. Section 4.2, section 4.3 and section 4.4 describe corpora, documents and annotations on documents respectively. Section 4.5 describes GATE's support for diverse document formats, and section 4.6 describes facilities for XML input/output.

## 4.1 Features: Simple Attribute/Value Data

GATE has a single model for information that describes documents, collections of documents (corpora), and annotations on documents, based on attribute/value pairs. Attribute names are strings; values can be any Java object. The API for accessing this feature data is Java's `Map` interface (part of the Collections API).

## 4.2 Corpora: Sets of Documents plus Features

A Corpus in GATE is a Java Set whose members are Documents. Both Corpora and Documents are types of LanguageResource (LR); all LRs have a FeatureMap (a Java Map) associated with them that stored attribute/value information about the resource. FeatureMaps are also used to associate arbitrary information with ranges of documents (e.g. pieces of text) via the annotation model (see below).

Documents have a DocumentContent which is a text at present (future versions may add support for audiovisual content) and one or more AnnotationSets which are Java Sets.

## 4.3 Documents: Content plus Annotations plus Features

Documents are modelled as content plus annotations (see section 4.4) plus features (see section 4.1). The content of a document can be any subclass of `DocumentContent`.

## 4.4 Annotations: Directed Acyclic Graphs

Annotations are organised in graphs, which are modelled as Java sets of Annotation. Annotations may be considered as the arcs in the graph; they have a start Node and an end Node, an ID, a type and a FeatureMap. Nodes have pointers into the sources document, e.g. character offsets.

### 4.4.1 Annotation Schemas

Annotation schemas provide a means to define types of annotations in GATE. GATE uses the XML Schema language supported by W3C for these definitions. When using the development environment to create/edit annotations, a component is available (`gate.gui.SchemaAnnotationEditor`) which is driven by an annotation schema file. This component will constrain the data entry process to ensure that only annotations that correspond to a particular schema are created. (Another component allows unrestricted annotations to be created.)

Schemas are resources just like other GATE components. Below we give some examples of such schemas. Section 2.22 describes how to create new schemas.

```
//////////////////////
// Date schema
//////////////////////
<?xml version="1.0"?>
<schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
 <!-- XSchema deffinition for Date-->
  <element name="Date">
    <complexType>
      <attribute name="kind"  use="optional">
        <simpleType>
          <restriction base="string">
            <enumeration value="date"/>
            <enumeration value="time"/>
            <enumeration value="dateTime"/>
          </restriction>
        </simpleType>
    </attribute>
  </complexType>
 </element>
</schema>

//////////////////////
// Person schema
//////////////////////
<?xml version="1.0"?>
<schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
    <!-- XSchema definition for Person-->
    <element name="Person" />
</schema>

//////////////////////////
// Address schema
//////////////////////////
<?xml version="1.0"?> <schema
xmlns="http://www.w3.org/2000/10/XMLSchema">
    <!-- XSchema deffinition for Address-->
    <element name="Address">
      <complexType>
        <attribute name="kind"  use="optional">
          <simpleType>
            <restriction base="string">
              <enumeration value="email"/>
              <enumeration value="url"/>
              <enumeration value="phone"/>
              <enumeration value="ip"/>
```

```
              <enumeration value="street"/>
              <enumeration value="postcode"/>
              <enumeration value="country"/>
              <enumeration value="complete"/>
            </restriction>
          </simpleType>
      </attribute>
  </complexType>
</element>
</schema>
```

### 4.4.2 Examples of Annotated Documents

This section shows some simple examples of annotated documents.

This material is adapted from [Grishman 97], the TIPSTER Architecture Design document upon which GATE version 1 was based. Version 2 has a similar model, although annotations are now graphs, and instead of multiple spans per annotation each annotation now has a single start/end node pair. The current model is largely compatible with [Bird & Liberman 99], and roughly isomorphic with "stand-off markup" as latterly adopted by the SGML/XML community.

Each example is shown in the form of a table. At the top of the table is the document being annotated; immediately below the line with the document is a ruler showing the position (byte offset) of each character. (**NOTE:** the ruler doesn't scale very well in HTML; for a better picture see the original TIPSTER Architecture Design Document[1].

Underneath this appear the annotations, one annotation per line. For each annotation is shown its Id, Type, Span (start/end offsets derived from the start/end nodes), and Features. Integers are used as the annotation Ids. The features are shown in the form name = value.

The first example shows a single sentence and the result of three annotation procedures: tokenization with part-of-speech assignment, name recognition, and sentence boundary recognition. Each token has a single feature, its part of speech (pos), using the tag set from the University of Pennsylvania Tree Bank; each name also has a single feature, indicating the type of name: person, company, etc.

Annotations will typically be organized to describe a hierarchical decomposition of a text. A simple illustration would be the decomposition of a sentence into tokens. A more complex case would be a full syntactic analysis, in which a sentence is decomposed into a noun phrase and a verb phrase, a verb phrase into a verb and its complement, etc. down to the level of individual tokens. Such decompositions can be represented by annotations on nested sets

[1] `http://www.itl.nist.gov/iaui/894.02/related_projects/tipster/`

| **Text** | | | | |
|---|---|---|---|---|
| **Cyndi savored the soup.** | | | | |
| **\|0...\|5...\|10..\|15..\|20** | | | | |
| **Annotations** | | | | |
| Id | Type | SpanStart | Span End | Features |
| 1 | token | 0 | 5 | pos=NP |
| 2 | token | 6 | 13 | pos=VBD |
| 3 | token | 14 | 17 | pos=DT |
| 4 | token | 18 | 22 | pos=NN |
| 5 | token | 22 | 23 | |
| 6 | name | 0 | 5 | name_type=person |
| 7 | sentence | 0 | 23 | |

Table 4.1: Result of annotation on a single sentence

of spans. Both of these are illustrated in the second example, which is an elaboration of our first example to include parse information. Each non-terminal node in the parse tree is represented by an annotation of type parse.

In most cases, the hierarchical structure could be recovered from the spans. However, it may be desirable to record this structure directly through a constituents feature whose value is a sequence of annotations representing the immediate constituents of the initial annotation. For the annotations of type parse, the constituents are either non-terminals (other annotations in the parse group) or tokens. For the sentence annotation, the constituents feature points to the constituent tokens. A reference to another annotation is represented in the table as "[ Annotation Id]"; for example, "[3]" represents a reference to annotation 3. Where the value of an feature is a sequence ofitems, these items are separated by commas. No special operations are provided in the current architecture for manipulating constituents. At a less esoteric level, annotations can be used to record the overall structure of documents, including in particular documents which have structured headers, as is shown in the third example (Table 4.3).

If the Addressee, Source, ... annotations are recorded when the document is indexed for retrieval, it will be possible to perform retrieval selectively on information in particular fields. Our final example (Table 4.4) involves an annotation which effectively modifies the document. The current architecture does not make any specific provision for the modification of the original text. However, some allowance must be made for processes such as spelling correction. This information will be recorded as a correction feature on token annotations and possibly on name annotations:

| **Text** | | | | |
|---|---|---|---|---|
| **Cyndi savored the soup.** | | | | |
| **\|0...\|5...\|10..\|15..\|20** | | | | |
| **Annotations** | | | | |
| Id | Type | SpanStart | Span End | Features |
| 1 | token | 0 | 5 | pos=NP |
| 2 | token | 6 | 13 | pos=VBD |
| 3 | token | 14 | 17 | pos=DT |
| 4 | token | 18 | 22 | pos=NN |
| 5 | token | 22 | 23 | |
| 6 | name | 0 | 5 | name_type=person |
| 7 | sentence | 0 | 23 | constituents=[1],[2],[3].[4],[5] |

Table 4.2: Result of annotations including parse information

| **Text** | | | | |
|---|---|---|---|---|
| **To: All Barnyard Animals** | | | | |
| **\|0...\|5...\|10..\|15..\|20** | | | | |
| **From: Chicken Little** | | | | |
| **\|25...\|30...\|35..\|40..\|45** | | | | |
| **Date: November 10,1194** | | | | |
| **\|50...\|55...\|60..\|65..** | | | | |
| **Subject: Descending Firmament** | | | | |
| **\|70...\|75...\|80..\|85..\|90..\|95..** | | | | |
| **Priority : Urgent.** | | | | |
| **\|100...\|105...\|110..\|115..** | | | | |
| **The sky is falling. The sky is falling.** | | | | |
| **\|120...\|125...\|130..\|135..\|140...\|145..\|150.** | | | | |
| **Annotations** | | | | |
| Id | Type | SpanStart | Span End | Features |
| 1 | Addressee | 4 | 24 | |
| 2 | Source | 31 | 45 | |
| 3 | Date | 53 | 69 | ddmmyy=101194 |
| 4 | Subject | 78 | 98 | |
| 5 | Priority | 109 | 115 | |
| 6 | Body | 116 | 155 | |
| 7 | Sentence | 116 | 135 | |
| 8 | Sentence | 136 | 155 | |

Table 4.3: Annotation showing overall document structure

| **Text** | | | | |
|---|---|---|---|---|
| **Topster tackles 2 terrorbytes.** | | | | |
| **\|0...\|5...\|10..\|15..\|20..\|25..** | | | | |
| **Annotations** | | | | |
| Id | Type | SpanStart | Span End | Features |
| 1 | token | 0 | 7 | pos=NP correction=TIPSTER |
| 2 | token | 8 | 15 | pos=VBZ |
| 3 | token | 16 | 17 | pos=CD |
| 4 | token | 18 | 29 | pos=NNS correction=terabytes |
| 5 | token | 29 | 30 | |

Table 4.4: Annotation modifying the document

### 4.4.3 Viewing and Editing Diverse Annotation Types

To view and edit annotation types, see Section 2.12. To add annotations of a new type, see Section 2.14. To add a new annotation schema, see Section 2.22.

## 4.5 Document Formats

The following document formats are supported by GATE:

- Plain Text
- HTML
- SGML
- XML
- RTF
- Email

By default GATE will try and identify the type of the document, then strip and convert any markup into GATE's annotation format. To disable this process, set the `markupAware` parameter on the document to `false`.

When reading a document of one of these types, GATE extracts the text between tags (where such exist) and create a GATE annotation filled as follows:

The name of the tag will constitute the annotation's type, all the tags attributes will materialize in the annotation's features and the annotation will span over the text covered by the tag. A few exceptions of this rule apply for the RTF, Email and Plain Text formats, which will be described later in the input section of these formats.

The text between tags is extracted and appended to the GATE document's content and all annotations created from tags will be placed into a GATE annotation set named "Original markups".

*Example:*

If the markup is like this:

```
<aTagName attrib1="value1" attrib2="value2" attrib3="value3"> A
piece of text</aTagName>
```

then the annotation created by GATE will look like:

```
annotation.type = "aTagName";
annotation.fm={attrib1=value1;atrtrib2=value2;attrib3=value3};
annotation.start=startNode;
annotation.end = endNode;
```

The startNode and endNode are created from offsets refereing the beginning and the end of "A piece of text" in the document's content.

The documents supported by GATE have to be in one of the encodings accepted by Java. The most popular is the *"UTF-8"* encoding which is also the most storage efficient one for UNICODE. If, when loading a document in GATE the *encoding* parameter is set to ""(the empty string), then the default encoding of the platform will be used.

### 4.5.1 Detecting the right reader

When opening a document in GATE, the file extension (e.g. `xml`) is important but if not present, GATE uses some other means to detect its type. In order to successfully apply the document creation algorithm described above, GATE needs to detect the proper reader to use for each document format. In order to do that, it takes into consideration (where possible) the information provided by three sources:

- Document's extension
- The web server's content type

- Magic numbers detection

The first represents the extension of a file like (*xml,htm,html,txt,sgm,rtf, etc*), the second represents the HTTP information sent by a web server regarding the content type of the document being send by it (*text/html; text/xml, etc*), and the third one represents certain sequences of chars which are ultimately number sequences. GATE is capable to support multimedia documents, if the right reader is added to the framework. Sometimes, multimedia documents are identified by a signature consisting in a sequence of numbers. Inside GATE they are called magic numbers. For textual documents, certain char sequences form such magic numbers. Examples of magic numbers sequences will be provided in the Input section of each format supported by GATE.

All those tests are applied to each document read, and after that, a voting mechanism decides what is the best reader to associate with the document. There is a degree of priority for all those tests. The document's extension test has the highest priority. If the system is in doubt which reader to choose, then the one associated with document's extension will be selected. The next higher priority is given to the web server's content type and the third one is given to the magic numbers detection. However, any two tests that identify the same mime type, will have the highest priority in deciding the reader that will be used. The web server test is not always successful as there might be documents that are loaded from a local file system, and the magic number detection test is not always applicable. In the next paragraphs we will se how those tests are performed and what is the general mechanism behind reader detection.

The method that detects the proper reader is a static one, and it belongs to the `gate.DocumentFormat` class. It uses the information stored in the maps filled by the init() method of each reader. This method comes with three signatures:

```
static public DocumentFormat getDocumentFormat( gate.Document
aGateDocument, URL url)

static public DocumentFormat getDocumentFormat(gate.Document
aGateDocument, String fileSuffix)

static public DocumentFormat getDocumentFormat(gate.Document
aGateDocument, MimeType mimeType)
```

The first two methods try to detect the right MimeType for the GATE document, and after that, they call the third one to return the reader associate with a MimeType. GATE uses the implementation from "http://jigsaw.w3.org" for mime types.

The magic numbers test is performed using the information form magic2mimeTypeMap map. Each key from this map, is searched in the first bufferSize (the

default value is 2048) chars of text. The method that does this is called `runMagicNumbers(InputStreamReader aReader)` and it belongs to DocumentFormat class. More details about it can be found in the GATE API documentation.

In order to activate a reader to perform the unpacking, the creole definition of a GATE document defines a parameter called "markupAware" initialized with a default value of **true**. This parameter, forces GATE to detect a proper reader for the document being read. If no reader is found, the document's content is load and presented to the user, just like any other text editor (this for textual documents).

The next subsections investigates particularities for each format and will describe the file extensions registered with each document format.

### 4.5.2 XML

#### Input

GATE permits the processing of any XML document and offers support for XML namespaces. It benefits the power of Apache's Xerces parser and also makes use of Sun's JAXP layer. Changing the XML parser in GATE can be achieved by simply replacing the value of a Java system property ("javax.xml.parsers.SAXParserFactory").

GATE will accept any well formed XML document as input. Although it has the possibility to validate XML documents against DTDs it does not do so because the validating procedure is time consuming and in many cases it issues messages that are annoying for the user.

There is an open problem with the general approach of reading XML, HTML and SGML documents in GATE. As we previously said, the text covered by tags/elements is appended to the GATE document content and a GATE annotation refers to this particular span of text. When appending, in cases such as "`end.</P><P>Start`" it might happen to concatenate the ending word of the previous annotation with the beginning phrase of the annotation currently being created, resulting in a garbage input for GATE processing resources that operate at the text surface.

*Let's take another example in order to better understand the problem :*

```
<title>This is a title</title><p>This is a paragraph</p><a
href="#link">Here is an useful link</a>
```

When the markup is transformed to annotations, it is likely that the text from the document's content will be as follows:

```
This is a titleThis is a paragraphHere is an useful link
```

The annotations created will refer the right parts of the texts but for the GATE's processing resources like (tokenizer, gazetter, etc) which work on this text, this will be a major diaster. Therefore, in order to prevent this problem from happening, GATE checks if it's likely to join words and if this happens then it inserts a space between those words. So, the text will look like this after loaded in GATE:

```
This is a title This is a paragraph Here is an useful link
```

There are cases when these words are meant to be joined, but they are just a few. This is why it's an open problem.

The extensions associate with the XML reader are:

- xml
- xhtm
- xhtml

The web server content type associate with xml documents is: *text/xml.*

The magic numbers test searches inside the document for the XML(`<?xml version="1.0"`) signature. It is also able to detect if the XML document uses the semantic described in the GATE document format DTD (see section 4.5.2) or uses other semantics.

**Output**

GATE is capable to assure persistence for its resources. These layers of persistence are various and they span until database persistence. However, for some purposes, a light and simple level of persistence would be highly appreciated. The types of persistent storage used for Language Resources are:

- Databases (like Oracle);
- Java serialization;
- XML serialization.

We describe the latter case in here.

XML persistence doesn't necessarily preserve all the objects belonging to the annotations, documents or corpora. Their features can be of all kinds of objects, with various layers of nesting. For example, *lists containing lists containing maps, etc.* Serializing these arbitrary data types in XML is not a simple task; GATE does the best it can, and supports native Java types such as Integers and Booleans, but where complex data types are used, information may

be lost(the types will be converted into Strings). GATE provides a full serialization of certain types of features such as collections, strings and numbers. It is possible to serialize only those collections containing strings or numbers. The rest of other features are serialized using their string representation and when read back, they will be all strings instead of being the original objects. Consequences of this might be observed when performing evaluations(see the evaluation section).

When GATE outputs an XML document it may do so in one of two ways:

- When the original document that was imported into GATE was an XML document, GATE can dump that document back into XML (possibly with additional markup added);
- For all document formats, GATE can dump its internal representation of the document into XML.

In the former case, the XML output will be close to the original document. In the latter case, the format is a GATE-specific one which can be read back by the system to recreate all the information that GATE held internally for the document.

In order to understand why there are two types of XML serialization, one needs to understand the structure of a GATE document. GATE allows a graph of annotations that refer to parts of the text. Those annotations are grouped under annotation sets. Because of this structure, sometimes it is impossible to save a document as XML using tags that surround the text referred by the annotation, because tags crossover situations could appear (XML is essentially a tree-based model of information, whereas GATE uses graphs). Therefore, in order to preserve all annotations in a GATE document, a custom type of XML document was developed.

The problem of crossover tags appears with GATE's second option (the preserve format one), which is implemented at the cost of loosing certain annotations. The way it is applied in GATE is that it tries to restore the original markup and where it is possible, to add in the same manner annotations produced by GATE.

**How to access and make use of the two ways of XML serialization**

**Save As XML option**

This option is available in GATE's GUI in the pop up menu associate with each language resource (document or corpus). Saving a corpus as XML is done by calling save as XML on each document of the corpus. This option saves all the annotations of a document together their features(applying the restrictions previously discussed), using the GateDocument.dtd :

```
<!ELEMENT GateDocument (GateDocumentFeatures,
          TextWithNodes, (AnnotationSet+))>
```

```
<!ELEMENT GateDocumentFeatures (Feature+)>
<!ELEMENT Feature (Name, Value)>
<!ELEMENT Name (\#PCDATA)>
<!ELEMENT Value (\#PCDATA)>
<!ELEMENT TextWithNodes (\#PCDATA | Node)*>
<!ELEMENT AnnotationSet (Annotation*)>
<!ATTLIST AnnotationSet  Name CDATA \#IMPLIED>
<!ELEMENT Annotation (Feature*)>
<!ATTLIST Annotation  Type      CDATA \#REQUIRED
                      StartNode CDATA \#REQUIRED
                      EndNode   CDATA \#REQUIRED>
<!ELEMENT Node EMPTY>
<!ATTLIST Node id CDATA \#REQUIRED>
```

The document is saved under a name chosen by the user and it may have any extension. However, the recommended extension would be "xml".

Using GATE's API, this option is available by calling `gate.Document's toXml()` method. This method returns a string which is the XML representation of the document on which the method was called.

**Note:** It is recommended that the string representation to be saved on the file system using the UTF-8 encoding, as the first line of the string is : `<?xml version="1.0" encoding="UTF-8"?>`

*Example of such a GATE format document:*

```
<?xml version="1.0" encoding="UTF-8" ?>
<GateDocument>

<!-- The =document's features-->

<GateDocumentFeatures>
<Feature>
  <Name className="java.lang.String">MimeType</Name>
  <Value className="java.lang.String">text/plain</Value>
</Feature>
<Feature>
  <Name className="java.lang.String">gate.SourceURL</Name>
  <Value className="java.lang.String">file:/G:/tmp/example.txt</Value>
</Feature>
</GateDocumentFeatures>

<!-- The document content area with serialized nodes -->

<TextWithNodes>
```

```
<Node id="0"/>A TEENAGER <Node
id="11"/>yesterday<Node id="20"/> accused his parents of cruelty
by feeding him a daily diet of chips which sent his weight
ballooning to 22st at the age of l2<Node id="146"/>.<Node
id="147"/>
</TextWithNodes>

<!-- The default annotation set -->

<AnnotationSet>
<Annotation Type="Date" StartNode="11"
EndNode="20">
<Feature>
  <Name className="java.lang.String">rule2</Name>
  <Value className="java.lang.String">DateOnlyFinal</Value>
</Feature> <Feature>
  <Name className="java.lang.String">rule1</Name>
  <Value className="java.lang.String">GazDateWords</Value>
</Feature> <Feature>
  <Name className="java.lang.String">kind</Name>
  <Value className="java.lang.String">date</Value>
</Feature> </Annotation> <Annotation Type="Sentence" StartNode="0"
EndNode="147"> </Annotation> <Annotation Type="Split"
StartNode="146" EndNode="147"> <Feature>
  <Name className="java.lang.String">kind</Name>
  <Value className="java.lang.String">internal</Value>
</Feature> </Annotation> <Annotation Type="Lookup" StartNode="11"
EndNode="20"> <Feature>
  <Name className="java.lang.String">majorType</Name>
  <Value className="java.lang.String">date_key</Value>
</Feature> </Annotation>
</AnnotationSet>

<!-- Named annotation set -->

<AnnotationSet Name="Original markups" >
 <Annotation
Type="paragraph" StartNode="0" EndNode="147"> </Annotation>
</AnnotationSet>
</GateDocument>
```

**Note:** One must know that all features that are not collections containing numbers or strings or that are not numbers or strings are discarded. With this option, GATE does not preserve those features it cannot restore back.

**The preserve format option**

This option is available in the GATE GUI from the popup menu of the annotations table. If no annotation in this table is selected, then the option will restore the document's original markup. If certain annotations are selected, then the option will attempt to restore the original markup and insert all the selected ones. When an annotation violates the crossed over condition, that annotation is discarded and a message is issued by GATE.

This option makes possible to generate an XML document with tags surrounding the annotation's refereed text and feature saved as attributes. All features which are collections, strings or numbers are saved, and the others are discarded. However, when read back, only the attributes under the GATE namespace (see bellow) are reconstructed back different than the others. That is because GATE does not store in the XML document the information about the features class and for collections the class of the items. So, when read back all features will become strings, except those under the GATE namespace.

One will notice that all generated tags have an attribute called "gateId" under the namespace "http://www.gate.ac.uk". The attribute is used when the document is read back in GATE, in order to restore the annotation's old ID. This feature is needed because it works in close cooperation with another attribute under the same namespace, called "matches". This attribute indicates annotations/tags that refer the same entity[2]. They are under this namespace because GATE is sensitive to them and treats them differently then all other elements with their attributes which falls under the general reading algorithm described at the beginning of this section.

The "gateId" under GATE namespace is used to create an annotation which have as ID, the value indicated by this attribute. The "matches" attribute is used to create an ArrayList in which the items will be Integers, representing the ID of annotations that the current one matches.

*Example:*

If the text being processed is as follows:

```
<Person gate:gateId="23">John</Person> and <Person
gate:gateId="25" gate:matches="23;25;30">John Major</Person> are
the same person.
```

What GATE does when parses this text, is to create two annotations:

```
a1.type = "Person"
a1.ID=Integer(23)
a1.start=<the start offset of
John>
a1.end = <the end offset of John>
```

[2]It's not an XML entity but a information extraction named entity

```
a1.featureMap = {}

a2.type="Person"
a2.ID = Integer(25)
a2.start= <the start offset
of John Major>
a2.end = <the end offset of John Major>
a2.featureMap ={matches=[Integer(23); Integer(25); Integer(30)]}
```

Under GATE's API, this option is available by calling `gate.Document's toXml(Set aSetContainingAnnotations)` method. This method returns a string which is the XML representation of the document on which the method was called. If called with **null** as a parameter, then the method will attempt to restore only the original markup. If the parameter is a set that contains annotations, then each annotation is tested against the crossover restriction, and for those found to violate it, a warning will be issued and they will be discarded.

In the next subsections we will show how this options applies to the other formats supported by GATE.

### 4.5.3 HTML

#### Input

The parser used to access HTML documents is the one provided by Java. The documents are read and created in GATE the same way as the XML documents.

The extensions associate with the HTML reader are:

- htm
- html

The web server content type associate with html documents is: *text/html*.

The magic numbers test searches inside the document for the HTML(`<html`) signature.There are certain HTML documents that do not contain the HTML tag, so the magical numbers test might not hold.

There is a certain degree of customization for HTML documents in that GATE introduces new lines into the document's text content in order to obtain a readable form. The annotations will refer the pieces of text as described in the original document but there will be a few extra new line characters inserted.

After reading H1,H2,H3,H4,H5,H6,TR,CENTER,LI,BR tags, GATE will introduce a new line(NL) char into the text. After a TITLE tag it will introduce two NLs. With P tags, GATE will introduce one NL at the beginning of the paragraph and one at the end of the paragraph. All newly added NLs are not considered to be part of the text contained by the tag.

#### Output

The Save as XML option works exactly the same for all GATE's documents so there is no particular observation to be made for the HTML formats.

When attempting to preserve the original markup formatting, GATE will generate the document in xhtml. The html document will look the same with any browser after processed by GATE but it will be in another syntax.

### 4.5.4 SGML

#### Input

The SGML support in GATE is fairly light as there is no freely available Java SGML parser. GATE uses a light converter attempting to transform the input SGML file into a well formed XML. Because it does not make use of a DTD, the conversion might not be always good. It is advisable to perform a SGML2XML conversion outside the system(using some other specialized tools) before using the SGML document inside GATE.

The extensions associate with the SGML reader are:

- sgm
- sgml

The web server content type associate with xml documents is : *text/sgml*.

There is no magic numbers test for SGML.

#### Output

When attempting to preserve the original markup formatting, GATE will generate the document as XML because the real input of a SGML document inside GATE is an XML one.

### 4.5.5 Plain text

#### Input

When reading a plain text document, GATE attempts to detect its paragraphs and add "paragraph" annotations to the document's "Original markups" annotation set. It does that by detecting two consecutive NLs. The procedure works for both UNIX like or DOS like text files.

*Example:*

If the plain text read is as follows:

```
Paragraph 1. This text belongs to the first paragraph.

Paragraph 2. This text belongs to the second paragraph
```

then two "paragraph" type annotation will be created in the "Original markups" annotation set (refereing the first and second paragraphs ) with an empty feature map.

The extensions associate with the plain text reader are:

- txt
- text

The web server content type associate with plain text documents is: *text/plain.*

There is no magic numbers test for plain text.

#### Output

When attempting to preserve the original markup formatting, GATE will dump XML markup that surrounds the text refereed.

The procedure described above applies both for plain text and RTF documents.

### 4.5.6 RTF

#### Input

Accessing RTF documents is performed by using the Java's RTF editor kit. It only extracts the document's text content from the RTF document.

The extension associate with the RTF reader is *"rtf"*.

The web server content type associate with xml documents is : *text/rtf*.

The magic numbers test searches for {\\rtf1.

#### Output

Same as the plain tex output.

### 4.5.7 Email

#### Input

GATE is able to read email messages packed in one document (UNIX mailbox format). It detects multiple messages inside such documents and for each message it creates annotations for all the fields composing an e-mail, like date, from, to, subject, etc. The message's body is analyzed and a paragraph detection is performed (just like in the plain text case) . All annotation created have as type the name of the e-mail's fields and they are placed in the Original markup annotation set.

*Example:*

```
From someone@zzz.zzz.zzz Wed Sep  6 10:35:50 2000

Date: Wed, 6 Sep2000 10:35:49 +0100 (BST)

From: forename1 surname2 <someone1@yyy.yyy.xxx>

To: forename2 surname2 <someone2@ddd.dddd.dd.dd>

Subject: A subject

Message-ID: <Pine.SOL.3.91.1000906103251.26010A-100000@servername>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII

This text belongs to the e-mail body....

This is a paragraph in the body of the e-mail

This is another paragraph.
```

GATE attempts to detect lines such "*From someone@zzz.zzz.zzz Wed Sep 6 10:35:50 2000*" in the e-mail text. Those lines separate e-mail messages contained in one file. After that, for each field in the e-mail message annotation are created as follows:

The annotation type will be the name of the field, the feature map will be empty and the annotation will span from the end of the filed until the end of the line containing the e-mail field.

*Example:*

```
a1.type = "date" a1 spans between the two ^ ^. Date:^ Wed,
6Sep2000 10:35:49 +0100 (BST)^

a2.type = "from"; a2 spans between the two ^ ^. From:^ forename1
surname2 <someone1@yyy.yyy.xxx>^
```

The extensions associate with the email reader are:

- eml
- email
- mail

The web server content type associate with plain text documents is: *text/email.*

The magic numbers test searches for keywords like *Subject:*,etc.

#### Output

Same as plain text output.

## 4.6 XML Input/Output

Support for input from and output to XML is described in section 4.5.2. In short:

- GATE will read any well-formed XML document (it does not attempt to validate XML documents). Markup will by default be converted into native GATE format.
- GATE will write back into XML in one of two ways:

1. Preserving the original format and adding selected markup (for example to add the results of some language analysis process to the document).
2. In GATE's own XML serialisation format, which encodes all the data in a GATE Document (as far as this is possible within a tree-structured paradigm – for 100% non-lossy data storage use GATE's RDBMS or binary serialisation facilities – see section 3.7).

When using the GATE framework, object representations of XML documents such as `DOM` or `jDOM`, or query and transformation languages such as `X-Path` or `XSLT`, may be used in parallel with GATE's own Document representation (`gate.Document`) without conflicts.

# Chapter 5

# JAPE: Regular Expressions Over Annotations

> If Osama bin Laden did not exist, it would be necessary to invent him. For the past four years, his name has been invoked whenever a US president has sought to increase the defence budget or wriggle out of arms control treaties. He has been used to justify even President Bush's missile defence programme, though neither he nor his associates are known to possess anything approaching ballistic missile technology. Now he has become the personification of evil required to launch a crusade for good: the face behind the faceless terror.
>
> The closer you look, the weaker the case against Bin Laden becomes. While the terrorists who inflicted Tuesday's dreadful wound may have been inspired by him, there is, as yet, no evidence that they were instructed by him. Bin Laden's presumed guilt appears to rest on the supposition that he is the sort of man who would have done it. But his culpability is irrelevant: his usefulness to western governments lies in his power to terrify. When billions of pounds of military spending are at stake, rogue states and terrorist warlords become assets precisely because they are liabilities.
>
> *The need for dissent*, George Monbiot, The Guardian, Tuesday September 18, 2001.

This chapter describes JAPE – a Java Annotation Patterns Engine. JAPE provides finite state transduction over annotations based on regular expressions. JAPE is a version of CPSL – Common Pattern Specification Language[1].

A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The phases run sequentially and constitute a cascade of finite state transducers over

[1]A good description of the original version of this language is in Doug Appelt's TextPro manual[2]. Doug was a great help to us in implementing JAPE. Thanks Doug!

annotations. The left-hand-side (LHS) of the rules consist of an annotation pattern that may contain regular expression operators (e.g. `*, ?, +`). The right-hand-side (RHS) consists of annotation manipulation statements. Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements.

At the beginning of each grammar, several options can be set:

- Control - this defines the method of rule matching (see Section refsec:jape:priority
- Debug - when set to true, if the grammar is running in Appelt mode and there is more than one possible match, the conflicts will be displayed on the standard output. See also Section 5.3.

Input annotations must also be defined at the start of each grammar. If no annotations are defined, the default will be Token, SpaceToken and Lookup (i.e. only these annotations will be considered when attempting a match). See Section 5.5 for more details.

There are 3 main ways in which the pattern can be specified:

- specify a string of text, e.g. {Token.string == "of"}
- specify the attributes (and values) of a token (or any other annotation), e.g. {Token.kind == number}
- specify an annotation previously assigned from a gazetteer, tokeniser, or other module, e.g. {Lookup.minorType == month}

Macros can also be used in the LHS of rules. This means that instead of expressing the information in the rule, it is specified in a macro, which can then be called in the rule. The reason for this is simply to avoid having to repeat the same information in several rules. Macros can themselves be used inside other macros.

The same operators can be used as for the tokeniser rules, i.e.

```
|
*
?
+
```

The pattern description is followed by a label for the annotation. Usually this label would be the same as the attribute value which will be assigned to it, although since the label is only local to the rule, this is for reasons of convenience rather than necessity. It is also possible to have more than one pattern and corresponding label, provided that the pattern is enclosed in a set of round brackets.

The RHS of the rule contains information about the annotation. Information about the annotation is transferred from the LHS of the rule using the label just described, and annotated with the entity type (which follows it). Finally, attributes and their corresponding values are added to the annotation. Alternatively, the RHS of the rule can contain Java code to create or manipulate annotations.

In the simple example below, the pattern described will be awarded an annotation of type "Enamex" (because it is an entity name). This annotation will have the attribute "kind", with value "location", and the attribute "rule", with value "GazLocation". (The purpose of the "rule" attribute is simply to ease the process of manual rule validation).

```
Rule: GazLocation
(
{Lookup.majorType == location}
)
:location -->
 :location.Enamex = {kind="location", rule=GazLocation}
```

Grammar rules can essentially be of two types. The first type of rule involves no gazetteer lookup, but can be defined using a small set of possible formats. In general, these are fairly straightforward and offer little potential for ambiguity.

The second type of rules rely more heavily on the gazetteer lists, and cover a much wider range of possibilities. This not only means that many rules may be needed to describe all situations, but that there is a much greater potential for ambiguity. This leads to the necessity for rule ordering and prioritisation, as will be discussed below.

For example, a single rule is sufficient to identify an IP address, because there is only one basic format - a series of numbers, each set connected by a dot. The rule for this is given below[3]:

```
Rule: IPAddress
(
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
)
```

[3]We might be more specific and state the possible lengths of the number, but within the confines of this project we currently have no need to, because there is no ambiguity with anything else

```
:ipAddress -->
 :ipAddress.Address = {kind = "ipAddress"}
```

To identify a date or time, there are many possible variations, and so many rules are needed. For example, the same date information can appear in the following formats (amongst others):

```
Wed, 10/7/00
Wed, 10/July/00
Wed, 10 July, 2000
Wed 10th of July, 2000
Wed. July 10th, 2000
Wed 10 July 2000
```

Different types of date can also be expressed. For example, the following would also be classified as date entities:

```
the late '80s
Monday
St. Andrew's Day
99 BC
mid-November
1980-81
from March to April
```

This also means there is a much greater potential for ambiguity. For example, many of the months of the year can also be girls' Christian names (e.g. May, June). This means that contextual information may be needed to disambiguate them, or we may have to guess which is more likely, based on frequency. For example, while "Friday" could be a person's name (as in "Man Friday"), it is much more likely to be a day of the week.

## 5.1 Use of Context

Context can be dealt with in the grammar rules in the following way. The pattern to be annotated is always enclosed by a set of round brackets. If preceding context is to be included in the rule, this is placed before this set of brackets. This context is described in exactly the same way as the pattern to be matched. If context following the pattern needs to be included, it is placed after the label given to the annotation. Context is used where a pattern should only be recognised if it occurs in a certain situation, but the context itself does not form part of the pattern to be annotated.

For example, the following rule for Time (assuming an appropriate macro for "year") would mean that a year would only be recognised if it occurs preceded by the words "in" or "by":

```
Rule: YearContext1

({Token.string == "in"}|
 {Token.string == "by"}
)
(YEAR)
:date -->
 :date.Timex = {kind = "date", rule = "YearContext1"}
```

Similarly, the following rule (assuming an appropriate macro for "email") would mean that an email address would only be recognised if it occurred inside angled brackets (which would not themselves form part of the entity):

```
Rule: Emailaddress1
({Token.string == ``<''})
(
 (EMAIL)
)
:email
({Token.string == ``>''})
-->
 :email.Address= {kind = "email", rule = "Emailaddress1"}
```

## 5.2 Use of Priority

Each grammar has 3 possible control styles: "brill", "first" and "appelt". This is specified at the beginning of the grammar. The brill style means that when more than one rule matches the same region of the document, they are all fired. The result of this is that a segment of text could be allocated more than one entity type, and that no proiority ordering is necessary.

With the "first" style, a rule fires for the first match that's found. This makes it unappropiate for rules that end in "+" or "?" or "*". Once a match is found the rule if fired; it does not attempt to get a longer match (as the other two styles do).

With the appelt style, only one rule can be fired for the same region of text, according to a set of priority rules. Priority operates in the following way.

1. From all the rules that match a region of the document starting at some point X, the one which matches the longest region is fired.

2. If more than one rule matches the same region, the one with the highest priority is fired
3. If there is more than one rule with the same priority, the one defined earlier in the grammar is fired.

An optional priority declaration is associated with each rule, which should be a positive integer. The higher the number, the greater the priority. By default (if the priority declaration is missing) all rules have the priority -1 (i.e. the lowest priority).

For example, the following two rules for location could potentially match the same text.

```
Rule:  Location1
Priority: 25

(
 ({Lookup.majorType == loc_key, Lookup.minorType == pre}
  {SpaceToken})?
 {Lookup.majorType == location}
 ({SpaceToken}
  {Lookup.majorType == loc_key, Lookup.minorType == post})?
)
:locName -->
  :locName.Location = {kind = "location", rule = "Location1"}


Rule: GazLocation
Priority: 20
  (
({Lookup.majorType == location}):location
  )
  -->  :location.Name = {kind = "location", rule=GazLocation}
```

Assume we have the text "China sea", that "China" is defined in the gazetteer as "location", and that sea is defined as a "loc_key" of type "post". In this case, rule Location1 would apply, because it matches a longer region of text starting at the same point ("China sea", as opposed to just "China"). Now assume we just have the text "China". In this case, both rules could be fired, but the priority for Location1 is highest, so it will take precedence. In this case, since both rules produce the same annotation, so it is not so important which rule is fired, but this is not always the case.

One important point of which to be aware is that prioritisation only operates within a single grammar. Although we could make priority global by having all the rules in a single grammar, this is not ideal due to other considerations. Instead, we currently combine all the

rules for each entity type in a single grammar. An index file (main.jape) is used to define which grammars should be used, and in which order they should be fired.

## 5.3 Useful tricks

Although the JAPE language has some limitations as to how rules and patterns can be expressed, there are some useful tricks to overcome these problems.

- Using priority to resolve ambiguity. If the Appelt style of matching is selected, rule priority operates in the following way.

  1. Length of rule – a rule matching a longer pattern will fire first.
  2. Explicit priority declaration. Use the optional Priority function to assign a ranking. The higher the number, the higher the priority. If no priority is stated, the default is -1.
  3. Order of rules. In the case where the above two factors do not distinguish between two rules, the order in which the rules are stated applies. Rules stated first have higher priority.

  Because priority can only operate within a single grammar, this can be a problem for dealing with ambiguity issues. One solution to this is to create a temporary set of annotations in initial grammars, and then manipulate this temporary set in one or more later phases (for example, by converting temporary annotations from different phases into permanent annotations in a single final phase. See the default set of grammars for an example of this.

- Negative operator. A negative operator cannot be specified as such. One solution to this is to create a "negative rule" which has higher priority than the matching "positive rule". The style of matching must be Appelt for this to work. To create a negative rule, simply state on the LHS of the rule the pattern that should NOT be matched, and on the RHS do nothing. In this way, the positive rule cannot be fired if the negative pattern matches, and vice versa, which has the same end result as using a negative operator. A useful variation for developers is to create a dummy annotation on the RHS of the negative rule, rather than to do nothing, and to give the dummy annotation a rule feature. In this way, it is obvious that the negative rule has fired. Alternatively, use Java code on the RHS to print a message when the rule fires. An example of a matching negative and positive rule follows. Here, we want a rule which matches a surname followed by a comma and a set of initials. But we want to specify that the initials shouldn't have the POS category PRP (personal pronoun). So we specify a negative rule that will fire if the PRP category exists, thereby preventing the positive rule from firing.

```
Rule: NotPersonReverse
Priority: 20
// we don't want to match ''Jones, I''
(
 {Token.category == NNP}
 {Token.string == ","}
 {Token.category == PRP}
)
:foo
-->
{}

Rule:  PersonReverse
Priority: 5
// we want to match ``Jones, F.W.''

(
 {Token.category == NNP}
 {Token.string == ","}
 (INITIALS)?
)
:person -->
```

- Matching special characters. To specify a single or double quote as a string, precede it with a backslash, e.g.

  ```
  {Token.string=="\""}
  ```

  will match a double quote. For other special characters, such as "$", enclose it in double quotes, e.g.

  ```
  {Token.category == "PRP$"}
  ```

- Referring to previous annotations. An annotation generated in one phase can be referred to in a later phase, in exactly the same way as any other kind of annotation (by specifying the name of the annotation within curly braces). The features and values can be referred to or omitted, as with all other annotations. Make sure that if the Input specification is used in the grammar, that the annotation to be referred to is included in the list.

- Using context. Specify left or right context around a pattern by enclosing it in round brackets outside the round brackets of the pattern. In the example below, the context "in" must precede the location to be annotated. Only the location will be annotated, but it is important to remember that context is consumed by the rule, so it cannot be reused in another rule within the same phase. So, for example, right context cannot be used as left context for another rule.

```
Rule:InLoc1
// in PARIS
(
 {Token.string == "in"}
)
(
 {Lookup.majorType == location, Token.orth == allCaps}
)
:locName
```

- Debug. Add the following to the options at the top of the grammar.

  ```
  Options: control = appelt debug = true
  ```

- Avoid conflicts. If two possible ways of matching are found for the same text string, a conflict can arise. Normally this is handled by the priority mechanism (test length, rule priority and finally rule precedence). If all these are equal, Jape will simply choose a match at random and fire it. This leads ot non-deterministic behaviour, which should be avoided.

- Using Java code on the RHS. If you want to be flash, you can use any Java code you like on the RHS of the rule. This is useful for feature percolation (see below), for deleting previous annotations, measuring length of strings, and performing alternative operations depending on particular features of the annotation. See 5.4 for more details.

- Feature percolation. To copy features from previous annotations, where the value of the feature is unknown, some simple Java code can be used. See Section 5.4 for a more detailed explanation of this.

## 5.4 Using Java code in JAPE rules

The RHS of a JAPE rule can consist of any Java code. This is useful for removing temporary annotations and for percolating and manipulating features from previous annotations. In the example below

The first rule below shows a rule which matches a first person name, e.g. “Fred”, and adds a gender feature depending on the value of the minorType from the gazetteer list in which the name was found. We first get the bindings associated with the person label (i.e. the Lookup annotation). We then create a new annotation called “personAnn” which contains this annotation, and create a new FeatureMap to enable us to add features. Then we get the minorType features (and its value) from the personAnn annotation (in this case, the feature will be “gender” and the value will be “male”), and add this value to a new feature called “gender”. We create another feature “rule” with value “FirstName”. Finally, we add all the

features to a new annotation "FirstPerson" which attaches to the same nodes as the original "person" binding.

Note that inputAS and outputAS represent the input and output annotation set. Normally, these would be the same (by default when using ANNIE, these will be the "Default" annotation set). Since the user is at liberty to change the input and output annotation sets in the paramters of the JAPE transducer at runtime, it cannot be guaranteed that the input and output annotation sets will be the same, and therefore we must specify the annotation set we are referring to.

```
Rule: FirstName

(
 {Lookup.majorType == person_first}
):person
-->
{
gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();
features.put("gender", personAnn.getFeatures().get("minorType"));
features.put("rule", "FirstName");
outputAS.add(person.firstNode(), person.lastNode(), "FirstPerson",
features);
}
```

The second rule (contained in a subsequent grammar phase) makes use of annotations produced by the first rule described above. Instead of percolating the minorType from the annotation produced by the gazetteer lookup, this time it percolates the feature from the annotation produced by the previous grammar rule. So here it gets the "gender" feature value from the "FirstPerson" annotation, and adds it to a new feature (again called "gender" for convenience), which is added to the new annotation (in outputAS) "TempPerson". At the end of this rule, the existing input annotations (from inputAS) are removed because they are no longer needed. Note that in the previous rule, the existing annotations were not removed, because it is possible they might be needed later on in another grammar phase.

```
Rule: GazPersonFirst
(
 {FirstPerson}
)
:person
-->
{
```

```
gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();

features.put("gender", personAnn.getFeatures().get("gender"));
features.put("rule", "GazPersonFirst");
outputAS.add(person.firstNode(), person.lastNode(), "TempPerson",
features);
inputAS.removeAll(person);
}
```

## 5.5 Optimising for speed

The way in which grammars are designed can have a huge impact on the processing speed. Some simple tricks to keep the processing as fast as possible are:

- avoid the use of the * and + operators. Replace them with ? where possible. For example, instead of

  ```
  ({Token})*
  ```

  use

  ```
  ({Token})? ({Token})? ({Token})?
  ```

  if you can predict that you won't need to recognise a string of Tokens longer than 3.

- avoid specifying unnecessary elements such as SpaceTokens where you can. To do this, use the Input specification at the beginning of the grammar to stipulate the annotations that need to be considered. If no Input specification is used, all annotations will be considered (so, for example, you cannot match two tokens separated by a space unless you specify the SpaceToken in the pattern). If, however, you specify Tokens but not SpaceTokens in the Input, SpaceTokens do not have to be mentioned in the pattern to be recognised. If, for example, there is only one rule in a phase that requires SpaceTokens to be specified, it may be judicious to move that rule to a separate phase where the SpaceToken can be specified as Input.

# Chapter 6

# ANNIE: a Nearly-New Information Extraction System

> And so the time had passed predictably and soberly enough in work and routine chores, and the events of the previous night from first to last had faded; and only now that both their days' work was over, the child asleep and no further disturbance anticipated, did the shadowy figures from the masked ball, the melancholy stranger and the dominoes in red, revive; and those trivial encounters became magically and painfully interfused with the treacherous illusion of missed opportunities. Innocent yet ominous questions and vague ambiguous answers passed to and fro between them; and, as neither of them doubted the other's absolute candour, both felt the need for mild revenge. They exaggerated the extent to which their masked partners had attracted them, made fun of the jealous stirrings the other revealed, and lied dismissively about their own. Yet this light banter about the trivial adventures of the previous night led to more serious discussion of those hidden, scarcely admitted desires which are apt to raise dark and perilous storms even in the pureset, most transparent soul; and they talked about those secret regions for which they felt hardly any longing, yet towards which the irrational wings of fate might one day drive them, if only in their dreams. For however much they might belong to one another heart and soul, they knew last night was not the first time they had been stirred by a whiff of freedom, danger and adventure.
>
> *Dream Story*, Arthur Schnitzler, 1926 (pp. 4-5).

GATE was originally developed in the context of Information Extraction[1] (IE) R&D, and IE systems in many languages and shapes and sizes have been created using GATE with the IE components that have been distributed with it (see [Maynard *et al.* 00] for descriptions of some of these projects).

[1] `http://www.dcs.shef.ac.uk/~hamish/IE/`

The principal architects of the IE systems in GATE version 1 were Robert Gaizauskas[2] and Kevin Humphreys. This work lives on in the LaSIE system[3][4], and in a number of design ideas since taken up in the ANNIE system described in this chapter(developed by Hamish Cunningham, Kalina Bontcheva, Valentin Tablan and Diana Maynard).

ANNIE stands for 'A Nearly-New IE' system. ANNIE borrows some ideas from LaSIE (and a part-of-speech tagger from Mark Hepple[5]), and contains a number of new ones, including a reliance on finite state algorithms and the JAPE language (see chapter 5).

ANNIE components form a pipeline which appears in figure 6.1. ANNIE components are

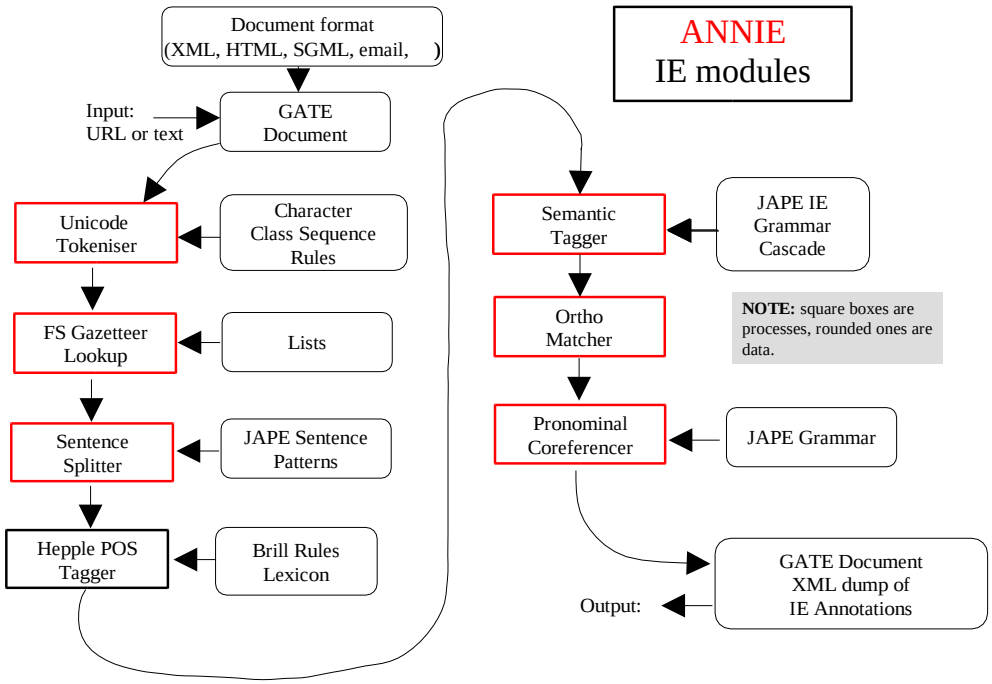



Figure 6.1: ANNIE and LaSIE

included with GATE (though the linguistic resources they rely on are generally more simple than the ones we use in-house). The rest of this chapter describes these components.

[2] `http://www.dcs.shef.ac.uk/ robertg`
[3] `http://nlp.shef.ac.uk/research/projects/all_projects.html`
[4] A derivative of LaSIE was distributed with GATE version 1 under the name VIE, a Vanilla IE system.
[5] `http://www.dcs.shef.ac.uk/ hepple`

## 6.1 Tokeniser

The tokeniser splits the text into very simple tokens such as numbers, punctuation and words of different types. For example, we distinguish between words in uppercase and lowercase, and between certain types of punctuation. The aim is to limit the work of the tokeniser to maximise efficiency, and enable greater flexibility by placing the burden on the grammar rules, which are more adaptable.

### 6.1.1 Tokeniser Rules

A rule has a left hand side (LHS) and a right hand side (RHS). The LHS is a regular expression which has to be matched on the input; the RHS describes the annotations to be added to the AnnotationSet. The LHS is separated from the RHS by '>'. The following operators can be used on the LHS:

```
| (or)
* (0 or more occurrences)
? (0 or 1 occurrences)
+ (1 or more occurrences)
```

The RHS uses ';' as a separator, and has the following format:

```
{LHS} > {Annotation type};{attribute1}={value1};...;{attribute
n}={value n}
```

Details about the primitive constructs available are given in the tokeniser file (DefaultTokeniser.Rules).

The following tokeniser rule is for a word beginning with a single capital letter:

```
"UPPERCASE_LETTER" "LOWERCASE_LETTER"* >
  Token;orth=upperInitial;kind=word;
```

It states that the sequence must begin with an uppercase letter, followed by zero or more lowercase letters. This sequence will then be annotated as type "Token". The attribute "orth" (orthography) has the value "upperInitial"; the attribute "kind" has the value "word".

### 6.1.2 Token Types

In the default set of rules, the following kinds of Token and SpaceToken are possible:

### Word

A word is defined as any set of contiguous upper or lowercase letters, including a hyphen (but no other forms of punctuation). A word also has the attribute "orth", for which four values are defined:

- upperInitial - initial letter is uppercase, rest are lowercase
- allCaps - all uppercase letters
- lowerCase - all lowercase letters
- mixedCaps - any mixture of upper and lowercase letters not included in the above categories

### Number

A number is defined as any combination of consecutive digits. There are no subdivisions of numbers.

### Symbol

Two types of symbol are defined: currency symbol (e.g. '$', '£') and symbol (e.g. '&', '^'). These are represented by any number of consecutive currency or other symbols (respectively).

### Punctuation

Three types of punctuation are defined: start_punctuation (e.g. '('), end_punctuation (e.g. ')'), and other punctuation (e.g. ':'). Each punctuation symbol is a separate token.

### SpaceToken

White spaces are divided into two types of SpaceToken - space and control - according to whether they are pure space characters or control characters. Any contiguous (and homogenous) set of space or control characters is defined as a SpaceToken.

The above description applies to the default tokeniser. However, alternative tokenisers can be created if necessary. The choice of tokeniser is then determined at the time of text processing.

## 6.2 Gazetteer

The gazetteer lists used are plain text files, with one entry per line. Each list represents a set of names, such as names of cities, organisations, days of the week, etc.

Below is a small section of the list for units of currency:

```
Ecu
European Currency Units
FFr
Fr
German mark
German marks
New Taiwan dollar
New Taiwan dollars
NT dollar
NT dollars
```

An index file (lists.def) is used to access these lists; for each list, a major type is specified and, optionally, a minor type [6]. In the example below, the first column refers to the list name, the second column to the major type, and the third to the minor type. These lists are compiled into finite state machines. Any text tokens that are matched by these machines will be annotated with features specifying the major and minor types. Grammar rules then specify the types to be identified in particular circumstances. Each gazetteer list should reside in the same directory as the index file.

```
currency_prefix.lst:currency_unit:pre_amount
currency_unit.lst:currency_unit:post_amount
date.lst:date:specific
day.lst:date:day
```

So, for example, if a specific day needs to be identified, the minor type "day" should be specified in the grammar, in order to match only information about specific days; if any kind of date needs to be identified,the major type "date" should be specified, to enable tokens annotated with any information about dates to be identified. More information about this can be found in the following section.

[6] it is also possible to include a language in the same way, where lists for different languages are used, though ANNIE is only concerned with monolingual recognition

## 6.3 Sentence Splitter

The **sentence splitter** is a cascade of finite-state transducers which segments the text into sentences. This module is required for the tagger. The splitter uses a gazetteer list of abbreviations to help distinguish sentence-marking full stops from other kinds.

Each sentence is annotated with the type Sentence. Each sentence break (such as a full stop) is also given a "Split" annotation. This has several possible types: ".", "punctuation", "CR" (a line break) or "multi" (a series of punctuation marks such as "?!?!".

The sentence splitter is domain and application-independent.

## 6.4 Part of Speech Tagger

The **tagger** [Hepple 00b] is a modified version of the Brill tagger, which produces a part-of-speech tag as an annotation on each word or symbol. The tagger uses a default lexicon and ruleset (the result of training on a large corpus taken from the Wall Street Journal). Both of these can be modified manually if necessary. Two additional lexicons exist - one for texts in all uppercase (lexicon_cap), and one for texts in all lowercase (lexicon_lower). To use these, the default lexicon should be replaced with the appropriate lexicon at load time. The default ruleset shoudl sitll be used in this case.

## 6.5 Semantic Tagger

ANNIE's semantic tagger is based on the JAPE language – see chapter refchap:jape. It contains rules which act on annotations assigned in earlier phases, in order to produce outputs of annotated entities.

## 6.6 Orthographic Coreference (OrthoMatcher)

(Note: this component was previously known as a "NameMatcher".)

The Orthomatcher module adds identity relations between named entities found by the semantic tagger, in order to perform coreference. It does not find new named entities as such, but it may assign a type to an unclassified proper name, using the type of a matching name.

The matching rules are only invoked if the names being compared are both of the same type, i.e. both already tagged as (say) organisations, or if one of them is classified as 'unknown'.

This prevents a previously classified name from being recategorised.

### 6.6.1 GATE Interface

Input – entity annotations, with an id attribute.

Output – matches attributes added to the existing entity annotations.

### 6.6.2 Resources

A lookup table is used to record non-matching strings which represent the same entity, e.g. "IBM" and "Big Blue", "Coca-Cola" and "Coke". The table is only read in at compile time, and since the module is tight coupled, the whole GATE executable must be recompiled to include any changes.

### 6.6.3 Processing

The wrapper builds an array of the strings, types and IDs of all `name` annotations, which is then passed to a string comparison function for pairwise comparisons of all entries.

## 6.7 Pronominal Coreference

The pronominal coreference module performs anaphora resolution using the JAPE grammar formalism. Note that this module is not automatically loaded with the other ANNIE modules, but can be loaded separately as a Processing Resource. The main module consists of three submodules:

- quoted text module
- pleonastic it module
- pronominal resolution module

The first two modules are helper submodules for the pronominal one, because they do not perform anything related to coreference resolution except the location of quoted fragments and pleonastic it occurrences in text. They generate temporary annotations which are used by the pronominal submodule (such temporary annotations are removed later).

The main coreference module can operate successfully only if all ANNIE modules were already executed. The module depends on the following annotations created from the respective ANNIE modules:

- Token (English Tokenizer)
- Sentence (Sentence Splitter)
- Split (Sentence Splitter)
- Location (NE Transducer, OrthoMatcher)
- Person (NE Transducer, OrthoMatcher)
- Organization (NE Transducer, OrthoMatcher)

For each pronoun (anaphor) the coreference module generates an annotation of type "Coreference" containing two features:

- antecedent offset - this is the offset of the starting node for the annotation (entity) which is proposed as the antecedent, or null if no antecedent can be proposed.
- matches - this is a list of annotation IDs that comprise the coreference chain comprising this anaphor/antecedent pair.

### 6.7.1 Quoted Speech Submodule

The quoted speech submodule identifies quoted fragments in the text being analysed. The identified fragments are used by the pronominal coreference submodule for the proper resolution of pronouns such as I, me, my, etc. which appear in quoted speech fragments. The module produces "Quoted Text" annotations.

The submodule itself is a JAPE transducer which loads a JAPE grammar and builds an FSM over it. The FSM is intended to match the quoted fragments and generate appropriate annotations that will be used later by the pronominal module.

The JAPE grammar consists of only four rules, which create temporary annotations for all punctuation marks that may enclose quoted speech, such as ", ', “, etc. These rules then try to identify fragments enclosed by such punctuation. Finally all temporary annotations generated during the processing, except the ones of type "Quoted Text", are removed (because no other module will need them later).

### 6.7.2 Pleonastic It submodule

The pleonastic it submodule matches pleonastic occurrences of "it". Similar to the quoted speech submodule, it is a JAPE transducer operating with a grammar containing patterns that match the most commonly observed pleonastic it constructs.

### 6.7.3 Pronominal Resolution Submodule

The main functionality of the coreference resolution module is in the pronominal resolution submodule. This uses the result from the execution of the quoted speech and pleonastic it submodules. The module works according to the following algorithm:

- Preprocess the current document. This step locates the annotations that the submodule need (such as Sentence, Token, Person, etc.) and prepares the appropriate data structures for them.
- For each pronoun do the following:
  - inspect the proper appropriate context for all candidate antecedents for this kind of pronoun;
  - choose the best antecedent (if any);
- Create the coreference chains from the individual anaphor/antecedent pairs and the coreference information supplied by the OrthoMatcher (this step is performed from the main coreference module).

### 6.7.4 Detailed description of the algorithm

Full details of the pronominal coreference algorithm are as dollows.

#### Preprocessing

The preprocessing task includes the following subtasks:

- Identifying the sentences in the document being processed. The sentences are identified with the help of the Sentence annotations generated from the Sentence Splitter. For each sentence a data structure is prepared that contains three lists. The lists contain the annotations for the person/organization/location named entities appearing in the sentence. The named entities in the sentence are identified with the help of the Person, Location and Organization annotations that are already generated from the Named Entity Transducer and the OrthoMatcher.

- The gender of each person in the sentence is identified and stored in a global data structure. It is possible that the gender information is missing for some entities - for example if only the person family name is observed then the Named Entity transducer will be unable to deduce the gender. In such cases the list with the matching entities generated by the OrhtoMatcher is inspected and if some of the orthographic matches contains gender information it is assigned to the entity being processed.
- The identified pleonastic it occurrences are stored in a separate list. The "Pleonastic It" annotations generated from the pleonastic submodule are used for the task.
- For each quoted text fragment, identified by the quoted text submodule, a special structure is created that contains the persons and the 3rd person singular pronouns such as "he" and "she" that appear in the sentence containing the quoted text, but not in the quoted text span (i.e. the ones preceding and succeeding the quote).

**Pronoun resolution**

This task includes the following subtasks:

etrieving all the pronouns in the document. Pronouns are represented as annotations of type "Token" with feature "category" having value "PRP" or "PRP$". The former classifies possessive adjectives such as my, your, etc. and the latter classifies personal, reflexive etc. pronouns. The two types of pronouns are combined in one list and sorted according to their offset in the text.

For each pronoun in the list the following actions are performed:

- If the pronoun is it then a check is performed if this is a pleonastic occurrence and if so then no further attempt for resolution is made.
- The proper context is determined. The context size is expressed in the number of sentences it will contain. The context always includes the current sentence (the one containing the pronoun), the preceding sentence and zero or more preceding sentences.
- Depending on the type of pronoun, a set of candidate antecedents is proposed. The candidate set includes the named entities that are compatible with this pronoun. For example if the current pronoun is she then only the Person annotations with "gender" feature equal to "female" or "unknown" will be considered as candidates.
- From all candidates, one is chosen according to evaluation criteria specific for the pronoun.

**Coreference chain generation**

This step is actually performed by the main module. After executing each of the submodules on the current document, the coreference module follows the steps:

- Retrieves the anaphor/antecedent pairs generated from them.
- For each pair, the orthographic matches (if any) of the antecedent entity is retrieved and then extended with the anaphor of the pair (i.e. the pronoun). The result is the coreference chain for the entity. The coreference chain contains the IDs of the annotations (entities) that co-refer.
- A new Coreference annotation is created for each chain. The annotation contains a single feature "matches" which value is the coreference chain (the list with IDs). The annotations are exported in a pre-specified annotation set.

The resolution for she, her, her$, he, him, his, herself and himself is similar because the analysis of the corpus showed that these pronouns are related to their antecedents in similar manner. The characteristics of the resolution process are:

- Context inspected is not very big - cases where the antecedent is found more than 3 sentences further back than the anaphor are rare.
- Recency factor is heavily used - the candidate antecedents that appear closer to the anaphor in the text are scored better.
- Anaphora have higher priority than cataphora. If there is an anaphoric candidate and a cataphoric one, then the anaphoric one is preferred, even if the recency factor scores the cataphoric candidate better.

The resolution process performs the following steps:

- Inspect the context of the anaphor for candidate antecedents. A candidate is considered every Person annotation. Cases where she/her refers to inanimate entity (ship for example) are not handled.
- For each candidate perform a gender compatibility check - only candidates having "gender" feature equal to "unknown" or compatible with the pronoun are considered for further evaluation.
- Evaluate each candidate with the best candidate so far. If the two candidates are anaphoric for the pronoun then choose the one that appears closer. The same holds for the case where the two candidates are cataphoric relative to the pronoun. If one is anaphoric and the other is cataphoric then choose the former, even if the latter appears closer to the pronoun.

**Resolution of it, its, itself**

This set of pronouns also shares many common characteristics. The resolution process contains certain differences with the one for the previous set of pronouns. Successful resolution for it, its, itself is more difficult because of the following factors:

- There is no gender compatibility restriction. In the case when there are several candidates in the context, the gender compatibility restriction is very useful for rejecting some of the candidates. When no such restriction exists, and with the lack of any syntactic or ontological information about the entities in the context, the recency factor plays the major role for choosing the best antecedent.
- The number of nominal antecedents (i.e. entities that are referred not by name) is much higher compared to the number of such antecedents for she, he, etc. In this case trying to find antecedent only amongst named entities degrades the precision a lot.

**Resolution of I, me, my, myself**

Resolution of these pronouns is dependent on the work of the quoted speech submodule. One important difference from the resolution process of other pronouns is that the context is not measured in sentences but depends solely on the quote span. Another difference is that the context is not contiguous - the quoted fragment itself is excluded from the context, because it is unlikely that an antecedent for I, me, etc. appears there. The context itself consists of:

- the part of the sentence where the quoted fragment originates, that is not contained in the quote - i.e. the text prior to the quote;
- the part of the sentence where the quoted fragment ends, that is not contained in the quote - i.e. the text following the quote;
- the part of the sentence preceding the sentence where the quote originates, which is not included in other quote.

It is worth noting that contrary to other pronouns, the antecedent for I, me, my and myself is most often cataphoric or if anaphoric it is not in the same sentence with the quoted fragment.

The resolution algorithm consists of the following steps:

- Locate the quoted fragment description that contains the pronoun. If the pronoun is not contained in any fragment then return without proposing an antecedent.

- Inspect the context for the quoted fragment (as defined above) for candidate antecedents. Candidates are considered annotations of type Pronoun or annotations of type Token with features category = "PRP", string = "she" or category = "PRP", string = "he".
- Try to locate a candidate in the text succeeding the quoted fragment (first pattern). If more than one candidate is present, choose the closest to the end of the quote. If a candidate is found then propose it as antecedent and exit.
- Try to locate candidate in the text preceding the quoted fragment (third pattern). Choose the closes one to the beginning of the quote. If found then set as antecedent and exit.
- Try to locate antecedents in the unquoted part of the sentence preceding the sentence where the quote starts (second pattern). Give preference to the one closest to the end of the quote (if any) in the preceding sentence or closest to the sentence beginning.

## 6.8 A Walk-Through Example

Let us take an example of a 3-stage procedure using the tokeniser, gazetteer and named-entity grammar. Suppose we wish to recognise the phrase "800,000 US dollars" as an entity of type "Number", with the feature "money".

First of all, we give an example of a grammar rule (and corresponding macros) for money, which would recognise this type of pattern.

```
Macro: MILLION_BILLION
({Token.string == "m"}|
{Token.string == "million"}|
{Token.string == "b"}|
{Token.string == "billion"}
)

Macro: AMOUNT_NUMBER
({Token.kind == number}
(({Token.string == ","}|
  {Token.string == "."})
{Token.kind == number})*
((SpaceToken.kind == space)?
 (MILLION_BILLION)?)
)
```

```
Rule: Money1
// e.g. 30 pounds
  (
      (AMOUNT_NUMBER)
      (SpaceToken.kind == space)?
      ({Lookup.majorType == currency_unit})
  )
 :money  -->
  :money.Number = {kind = "money", rule = "Money1"}
```

### 6.8.1 Step 1 - Tokenisation

The tokeniser separates this phrase into the following tokens. In general, a word is comprised of any number of letters of either case, including a hyphen, but nothing else; a number is composed of any sequence of digits; punctuation is recognised individually (each character is a separate token), and any number of consecutive spaces and/or control characters are recognised as a single spacetoken.

```
Token, string = ‘‘800’’, kind = number, length = 3
Token, string = ‘‘,’’, kind = punctuation, length = 1
Token, string = ‘‘000’’, kind = number, length = 3
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘US’’, kind = word, length = 2, orth = allCaps
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘dollars’’, kind = word, length = 7, orth = lowercase
```

### 6.8.2 Step 2 - List Lookup

The gazetteer lists are then searched to find all occurrences of matching words in the text. It finds the following match for the string “US dollars”:

```
Lookup, minorType = post_amount, majorType = currency_unit
```

### 6.8.3 Step 3 - Grammar Rules

The grammar rule for money is then invoked. The macro MILLION_BILLION recognises any of the strings “m”, “million”, “b”, “billion”. Since none of these exist in the text, it passes onto the next macro. The AMOUNT_NUMBER macro recognises a number, optionally followed by any number of sequences of the form“dot or comma plus number”, followed

by an optional space and an optional MILLION_BILLION. In this case, "800,000" will be recognised. Finally, the rule Money1 is invoked. This recognises the string identified by the AMOUNT_NUMBER macro, followed by an optional space, followed by a unit of currency (as determined by the gazetteer). In this case, "US dollars" has been identified as a currency unit, so the rule Money1 recognises the entire string "800,000 US dollars". Following the rule, it will be annotated as a Number entity of type Money:

```
Number, kind = money, rule = Money1
```

# Chapter 7

# More CREOLE

> For the previous reader was none other than myself. I had already read this book long ago.
>
> The old sickness has me in its grip again: amnesia in litteris, the total loss of literary memory. I am overcome by a wave of resignation at the vanity of all striving for knowledge, all striving of any kind. Why read at all? Why read this book a second time, since I know that very soon not even a shadow of a recollection will remain of it? Why do anything at all, when all things fall apart? Why live, when one must die? And I clap the lovely book shut, stand up, and slink back, vanquished, demolished, to place it again among the mass of anonymous and forgotten volumes lined up on the shelf.
>
> ...
>
> But perhaps - I think, to console myself - perhaps reading (like life) is not a matter of being shunted on to some track or abruptly off it. Maybe reading is an act by which consciousness is changed in such an imperceptible manner that the reader is not even aware of it. The reader suffering from amnesia in litteris is most definitely changed by his reading, but without noticing it, necause as he reads, those critical faculties of his brain that could tell him that change is occurring are changing as well. And for one who is himself a writer, the sickness may conceivably be a blessing, indeed a necessary precondition, since it protects him against that crippling awe which every great work of literature creates, and because it allows him to sustain a wholly uncomplicated relationship to plagiarism, without which nothing original can be created.
>
> *Three Stories and a Reflection*, Patrick Suskind, 1995 (pp. 82, 86).

This chapter describes additional CREOLE resources which do not form part of ANNIE.

## 7.1 Document Reset

The document reset resource enables the document to be reset to its original state, by removing all the annotation sets and their contents, apart from the one containing the document format analysis (Original Markups). This resource is normally added to the beginning of an application, so that a document is reset before an application is rerun on that document.

## 7.2 Verb Group Chunker

The rule-based verb chunker is based on a number of grammars of English [Cobuild 99, Azar 89]. We have developed 68 rules for the identification of non recursive verb groups. The rules cover finite ('is investigating'), non-finite ('to investigate'), participles ('investigated'), and special verb constructs ('is going to investigate'). All the forms may include adverbials and negatives. The rules have been implemented in JAPE. The finite state analyser produces an annotation of type 'VG' with features and values that encode syntactic information ('type', 'tense', 'voice', 'neg', etc.). The rules use the output of the POS tagger as well as information about the identity of the tokens (e.g. the token 'might' is used to identify modals).

The grammar for verb group identification can be loaded as a Jape grammar into the GATE architecture and can be used in any application: the module is domain independent.

## 7.3 Noun Group Chunker

The NG chunker is based on the algorithm used in the POS tagger (described in [Hepple 00a]). We have developed a transformation based chunking procedure that learns how to identify non-recursive noun chunks in a text. The algorithm is trained on the Wall Street Journal (WSJ) Corpus. The original parsed WSJ was re-tagged in order to include information about non-recursive noun group chunks.

Two sets of tags are being used for experimentation, shown in Table 7.1. Set I is very simple while Set II is richer because it encodes information about the head of the non-recursive noun group: this has a clear impact on system performance. In addition to the chunk tags, the corpus preserves the information about POS tags (the Brown Corpus Tag Set).

The rules for text chunking take into consideration word identity and POS information as features for learning.

The learning procedure is as follows:

- baseline system: from the corpus the number of times a given chunk tag occurs with

| Tag Set I | |
|---|---|
| Tag and Definition | Example |
| B: Begin Chunk<br>I: Inside Chunk<br>O: Outside Chunk | Dr./B Talcott/I led/O a/B team/I of/O researchers/B from/O ...<br>Chunks: "Dr. Talcott" "a team" "researchers" |
| **Tag Set II** | |
| Tag and Definition | Example |
| B: Begin Chunk<br>I: Inside Chunk<br>E: End Chunk<br>U: One Token Chunk<br>O: Outside Chunk | Dr./B Talcott/E led/O a/B team/E of/O researchers/U from/O ...<br>Chunks: "Dr. Talcott" "a team" "researchers" |

Table 7.1: Two Tag Sets Used for Transformation-based Chunking.

a given POS tag is computed. Given a word with POS tag *pos*, the baseline system assigns to the word the most probable chunk tag for *pos* (as observed in the training corpus);

- learning rules: rules are learnt that correct the errors made by the baseline system. The rules are learnt gradually for each position in the corpus where an error is observed in the baseline system. These rules have the form:

```
CHANGE <CHUNK_TAG_1> BY <CHUNK_TAG_2> IF
CONDITION ON WORDS AND
CONDITION ON POS TAGS AND
CONDITION ON CHUNK TAGS
```

In order to limit the number of possible rules to learn, templates are defined that take into consideration windows around the chunk to be corrected (the language bias). We consider three words to the left, three words to the right, and the word itself; three POS to the left, three POS to the right; and the POS itself; and two chunks to the left and two chunks to the right. So a possible rule is:

```
CHANGE I BY B IF
POS_1 = ‘‘IN’’
```

This rule specifies to change the current "I" chunk by a "B" chunk when the previous POS tag is "IN" (e.g., a preposition).

Rules are learnt until a user-specified threshold is reached (no improvement is observed).

In order to implement the chunker in GATE we have developed a pre-chunking procedure: a Java class that adds contextual information to each token. We have also developed code for the automatic translation of the baseline system and the rules into JAPE grammars, and we have defined a JAPE grammar that identifies chunks by means of regular expressions (e.g., B(I)*).

## 7.4 OntoText Gazetteer

The OntoText Gazetteer is a Natural Gazetteer, implemented from the OntoText Lab (http://www.ontotext.com/). Its implementaion is based on simple lookup in several java.util.HashMap, and is inspired by the strange idea of Atanas Kiryakov, that searching in HashMaps will be faster than a search in a Finite State Machine (FSM).

Here follows a description of the algorithm that lies behind this implementation:

Every phrase i.e. every list entry is separated into several parts. The parts are determined by the whitespaces lying among them. e.g. the phrase : "form is emptiness" has three parts : form, is & emptiness. There is also a list of HashMaps: mapsList which has as many elements as the longest (in terms of "count of parts") phrase in the lists. So the first part of a phrase is placed in the first map. The first part + space + second part is placed in the second map, etc. The full phrase is placed in the appropriate map, and a reference to a Lookup object is attached to it.

On first sight it seems that this algorithm is certainly much more memory-consuming than a finite state machine (FSM) with the parts of the phrases as transitions, but this is actually not so important since the average length of the phrases (in parts) in the lists is 1.1. On the other hand, one advantage of the algorithm is that, although unconventional, on average it takes four times less memory and works three times faster than an optimized FSM implementation.

The lookup part is implemented in execute() so a lot of tokenization takes place there. After defining the candidates for phrase-parts, we build a candidate phrase and try to look it up in the maps (in which map again depends on the count of parts in the current candidate phrase).

### 7.4.1 Prerequisites

The phrases to be recognised should be listed in a set of files, one for each type of occurrence (as for the standard gazetteer).

The gazetteer is built with the information from a file that contains the set of lists (which are files as well) and the associated type for each list. The file defining the set of lists should have the following syntax: each list definition should be written on its own line and should

contain:

- the file name (required)
- the major type (required)
- the minor type (optional)
- the language(s) (optional)

The elements of each definition are separated by ":". The following is an example of a valid definition:

```
personmale.lst:person:male:english
```

Each file named in the lists definition file is just a list containing one entry per line.

When this gazetter is run over some input text (a Gate document) it will generate annotations of type Lookup having the attributes specified in the definition file.

### 7.4.2 Setup

In order to use this gazetteer from within GATE the following should reside in the creole setup file (creole.xml):

```
    <RESOURCE>
      <NAME>OntoText Gazetteer</NAME>
      <CLASS>com.ontotext.gate.gazetteer.NaturalGazetteer</CLASS>
      <COMMENT>A list lookup component. for documentation please refer to
(www.ontotext.com/gate/gazetteer/documentation/index.html). For licence
information please refer to
(www.ontotext.com/gate/gazetteer/documentation/licence.ontotext.html) or to
licence.ontotext.html in the lib folder of
GATE</COMMENT>
      <PARAMETER NAME="document" RUNTIME="true" COMMENT="The document to be
processed">gate.Document</PARAMETER>
      <PARAMETER NAME="annotationSetName" RUNTIME="true" COMMENT="The
annotation set to be used for the generated
annotations" OPTIONAL="true">java.lang.String</PARAMETER>
      <PARAMETER NAME="listsURL"
DEFAULT="gate:/creole/gazeteer/default/lists.def" COMMENT="The URL to the
file with list of
```

```
lists" SUFFIXES="def">java.net.URL</PARAMETER>
      <PARAMETER DEFAULT="UTF-8" NAME="encoding" COMMENT="The encoding used
for reading the
definitions">java.lang.String</PARAMETER>
      <PARAMETER DEFAULT="true" NAME="caseSensitive" COMMENT="Should this
gazetteer diferentiate on case. Currently the
Gazetteer works only in case sensitive mode.">java.lang.Boolean</PARAMETER>
      <ICON>shefGazetteer.gif</ICON>
    </RESOURCE>
```

## 7.5 Flexible Exporter

The Flexible Exporter enables the user to save a document in its original format with added annotations. The user can select the name of the annotation set from which these annotations are to be found, which annotations from this set are to be included, whether features are to be included, and various renaming options such as renaming the annotations and the file.

For information on how to use the flexible exporter, see Section 2.9.2.

## 7.6 Annotation Set Transfer

The Annotation Set Transfer enables the parts of a document matching a particular annotation to be transferred into a new annotation set. For example, this can be used when a user only wants to run a processing resource over a specific part of a document, such as the Body of an HTML document. The user specifies the name of the annotation set and the annotation which covers the part of the document they wish to transfer, and the name of the new annotation set. All the other annotations corresponding to the matched text will be transferred to the new annotation set. For example, we might wish to perform named entity recognition on the body of an HTML text, but not on the headers. After tokenising and performing gazetteer lookup on the whole text, we would use the Annotation Set Transfer to transfer those annotations (created by the tokeniser and gazetteer) into a new annotation set, and then run the remaining NE resources, such as the semantic tagger and coreference modules, on them.

For more information about how to use the Annotation Set transfer, see Section 2.9.2.

# Chapter 8

# Performance Evaluation of Language Analysers

> When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science. (Kelvin)
>
> Not everything that counts can be counted, and not everything that can be counted counts. (Einstein)

GATE provides two useful tools for automatic evaluation: the AnnotationDiff tool and the Benchmarking Tool. These are particularly useful not just as a final measure of performance, but as a tool to aid system development by tracking progress and evaluating the impact of changes as they are made. The evaluation tool (AnnotationDiff) enables automated performance measurement and visualisation of the results, while the benchmarking tool enables the tracking of a system's progress and regression testing.

## 8.1 The AnnotationDiff Tool

The AnnotationDiff tool enables two sets of annotations on a document to be compared, in order either to compare a system-annotated text with a reference (hand-annotated) text, or to compare the output of two different versions of the system (or two different systems). For each annotation type, figures are generated for precision, recall, F-measure and false positives. Each of these can be calculated according to 3 different criteria - strict, lenient and average. The reason for this is to deal with partially correct responses in different ways.

- The Strict measure considers all partially correct responses as incorrect (spurious).

- The Lenient measure considers all partially correct responses as correct.
- The Average measure allocates a half weight to partially correct responses (i.e. it takes the average of strict and lenient).

It can be accessed both from GUI or from the API. Annotation Diff compares sets of annotations with the same type. When performing the diff, the annotation offsets and their features will be taken into consideration. and after that, the diff process is triggered. Figure 8.1 shows a part of the AnnotationDiff viewer.

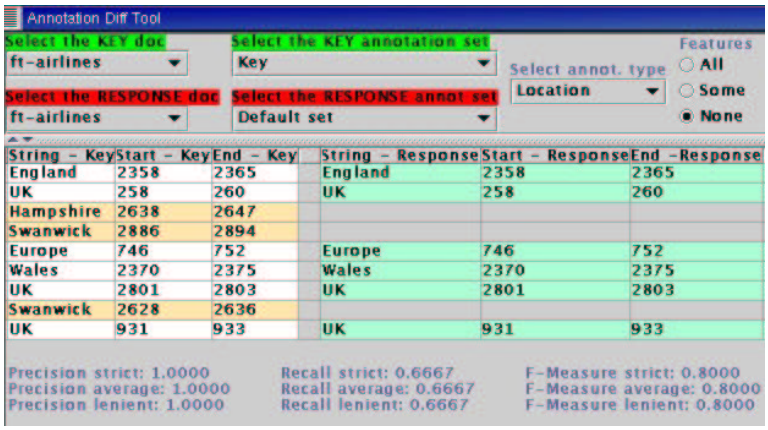



Figure 8.1: Part of the AnnotationDiff viewer

All annotations from the key set are compared with the ones from the response set, and those found to have the same start and end offsets are displayed on the same line in the table. Next, Annotation Diff evaluates if the features of each annotation from the response set subsume those features from the key set, as specified by the keyFeatureNamesSet parameter. To understand this in more detail, see section 2.17, which describes the Annotation Diff parameters.

## 8.2 The six annotation relations explained

### Coextensive

Two annotations are coextensive if they hit the same span of text in a document. Basically, both their start and end offsets are equal.

### Overlaps

Two annotations overlap if they share a common span of text.

**Compatible**

Two annotations are compatible if they are coextensive and if the features of one (usually the ones from the key) are included in the features of the other (usually the response).

**Partially Compatible**

Two annotations are partially compatible if they overlap and if the features of one (usually the ones from the key) are included in the features of the other (response).

**Missing** This applies only to the key annotations. A key annotation is missing if either it is not coextensive or overlapping, orif one or more features are not included in the response annotation.

**Spurious**

This applies only to the response annotations. A response annotation is spurious if either it is not coextensive or overlapping, or if one or more features from the key are not included in the response annotation.

## 8.3 Benchmarking tool

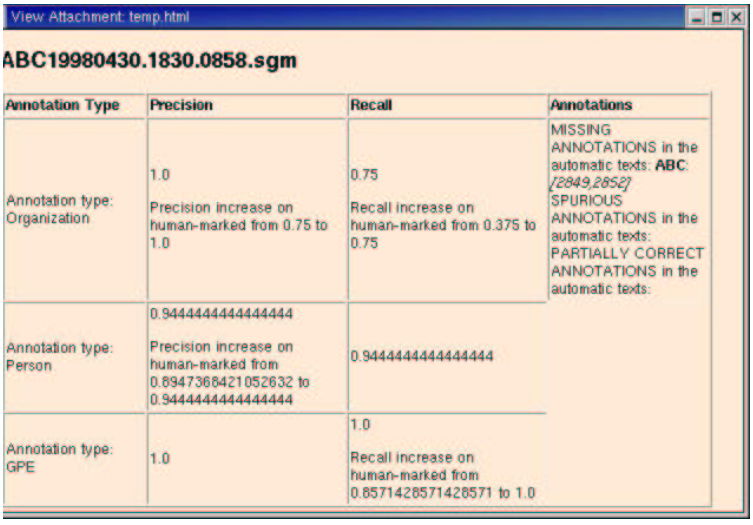
The benchmarking tool differs from the AnnotationDiff in that it enables evaluation to be carried out over a whole corpus rather than a single document. It also enables tracking of the system's performance over time. The tool can be run in either GUI mode or standalone mode. For more information on how to run the tool, see 2.18.

The tool requires a clean version of a corpus (with no annotations) and an annotated reference corpus. First of all, the tool is run in generation mode to produce a set of texts annotated by the system. These texts are stored for future use. The tool can then be run in three ways:

1. comparing the stored processed set with the human-annotated set;
2. comparing the current processed set with the human-annotated set;
3. (default mode) comparing the stored processed set with the current processed set and the human-annotated set.

In each case, performance statistics will be output for each text in the set, and overall statistics for the entire set. In the default mode, information is also provided about whether the figures have increased or decreased in comparison with the annotated set. The processed set can be updated at any time by rerunning the tool in generation mode with the latest version of the system resources. Furthermore, the system can be run in verbose mode, where for each P and R figure below a certain threshold (set by the user), the non-coextensive

annotations (and their corresponding text) will be displayed. The output of the tool is written to an HTML file in tabular form, for easy viewing of the results (see Figure 8.2).


View Attachment: temp.html

**ABC19980430.1830.0858.sgm**

| Annotation Type | Precision | Recall | Annotations |
|---|---|---|---|
| Annotation type: Organization | 1.0<br>Precision increase on human-marked from 0.75 to 1.0 | 0.75<br>Recall increase on human-marked from 0.375 to 0.75 | MISSING ANNOTATIONS in the automatic texts: **ABC** *[2849,2852]*<br>SPURIOUS ANNOTATIONS in the automatic texts:<br>PARTIALLY CORRECT ANNOTATIONS in the automatic texts: |
| Annotation type: Person | 0.9444444444444444<br>Precision increase on human-marked from 0.6947368421052632 to 0.9444444444444444 | 0.9444444444444444 | |
| Annotation type: GPE | 1.0 | 1.0<br>Recall increase on human-marked from 0.8571428571428571 to 1.0 | |

Figure 8.2: Fragment of results from benchmark tool

## 8.4 Metrics for Evaluation in Information Extraction

Much of the research in IE in the last decade has been connected with the MUC competitions, and so it is unsurprising that the MUC evaluation metrics of precision, recall and F-measure [Chinchor 92] also tend to be used, along with slight variations. These metrics have a very long-standing tradition in the field of IR [van Rijsbergen 79] (see also [Manning & Schütze 99, Frakes & Baeza-Yates 92]).

**Precision** measures the number of correctly identified items as a percentage of the number of items identified. In other words, it measures how many of the items that the system identified were actually correct, regardless of whether it also failed to retrieve correct items. The higher the precision, the better the system is at ensuring that what is identified is correct.

**Error rate** is the inverse of precision, and measures the number of incorrectly identified items as a percentage of the items identified. It is sometimes used as an alternative to precision.

**Recall** measures the number of correctly identified items as a percentage of the total number of correct items. In other words, it measures how mnay of the items that should have been identified actually were identified, regardless of how many spurious identifications were made.

The higher the recall rate, the better the system is at not missing correct items.

Clearly, there must be a tradeoff between precision and recall, for a system can easily be made to achieve 100% precision by identifying nothing (and so making no mistakes in what it identifies), or 100% recall by identifying everything (and so not missing anything). The **F-measure** [van Rijsbergen 79] is often used in conjunction with Precision and Recall, as a weighted average of the two. if the weight is set to 0.5, precision and recall are deemed equally important.

**False positives** are a useful metric when dealing with a wide variety of text types, because it is not dependent on *relative document richness*[1] in the same way that precision is.

When comparing different systems on the same document set, relative document richness is unimportant, because it is equal for all systems. When comparing a single system's performance on different documents, however, it is much more crucial, because if a particular document type has a significantly different number of any type of entity, the results for that entity type can become skewed. Compare the impact on precision of one error where the total number of correct entities = 1, and one error where the total = 100. Assuming the document length is the same, then the false positive score for each text, on the other hand, should be identical.

Common metrics for evaluation of IE systems are defined as follows:

$$Precision = \frac{Correct + 1/2Partial}{Correct + Spurious + 1/2Partial} \tag{8.1}$$

$$Recall = \frac{Correct + 1/2Partial}{Correct + Missing + 1/2Partial} \tag{8.2}$$

$$F - measure = \frac{(\beta^2 + 1)P * R}{(\beta^2 R) + P} \tag{8.3}$$

where $\beta$ is a value between 0 and 1 reflecting the weighting of P vs. R. If $\beta$ is set to 0.5, the two are weighted equally.

$$FalsePositive = \frac{Spurious}{c} \tag{8.4}$$

where $c$ is some constant independent from document richness, e.g. the number of tokens or sentences in the document.

Note that we consider annotations to be partially correct if the entity type is correct and the spans are overlapping but not identical. Partially correct responses are normally allocated a half weight.

[1] By this we mean the relative number of entities of each type to be found in a set of documents.

# Chapter 9

# Users, Groups, and LR Access Rights

"Well," he said, "it's to do with the project which first made the software incarnation of the company profitable. It was called Reason, and in its own way it was sensational."

"What was it?"

"Well, it was a kind of back-to-front program. It's funny how many of the best ideas are just an old idea back-to-front. You see there have already been several programs written that help you to arrive at decisions by properly ordering and analysing all the relevant facts so that they then point naturally towards the right decision. The drawback with these is that the decision which all the properly ordered and analysed facts point to is not necessarily the one you want."

"Yeeeess ..." said Reg's voice from the kitchen.

"Well, Gordon's great insight was to design a program which allowed you to specify in advance what decision you wished it to reach, and only then to give it all the facts. The program's task, which it was able to accomplish with consummate ease, was simply to construct a plausible series of logical-sounding steps to connect the premises with the conclusion.

"And I have to say that it worked brilliantly. Gordon was able to buy himself a Porsche almost immediately despite being completely broke and a hopeless driver. Even his bank manager was unable to find fault with his reasoning. Even when Gordon wrote it off three weeks later."

"Heavens. And did the program sell very well?"

"No. We never sold a single copy."

"You astonish me. It sounds like a real winner to me."

"It was," said Richard hesitantly. "The entire project was bought up, lock, stock and barrel, by the Pentagon. The deal put WayForward on a very sound financial foundation. Its moral foundation, on the other hand, is not something I would

> want to trust my weight to. I've recently been analysing a lot of the arguments put forward in favour of the Star Wars project, and if you know what you're looking for, the pattern of the algorithms is very clear.
>
> "So much so, in fact, that looking at Pentagon policies over the last couple of years I think I can be fairly sure that the US Navy is using version 2.00 of the program, while the Air Force for some reason only has the beta-test version of 1.5. Odd, that."
>
> *Dirk Gently's Holistic Detective Agency*, Douglas Adams, 1987 (pp. 55-56).

This chapter describes the LR access mechanism which is implemented for persistent LRs. At present there are two LR persistency storage methods: Java serialisation and Oracle. Here we will describe their security features in turn.

## 9.1 Java serialisation and LR access rights

At present the security model is not implemented for Java serialization. One should rely on the security control offered by the OS in order to restrict access to certain persistent resources.

## 9.2 Oracle Datastore and LR access rights

**Warning:** These features will not work, unless you have an Oracle pre-installed at your site[1] and you, or an administrator at your site, has installed the GATE Oracle support (see http://gate.ac.uk/gate/doc/persistence.pdf).

Oracle datastores have advanced LR access rights based on users and groups, which are similar to those in an operating system such as Linux.

In order to be able to access an LR stored in an Oracle datastore, a user needs to supply a user name, password and a group. These credentials are used to determine which LRs are accessible to this user for reading and writing.

### 9.2.1 Users, Groups, Sessions and Access Modes

The security model provides primitives such as users, groups, permissions and sessions similar to the ones provided by the operating systems:

[1] Oracle installation is not provided with Gate. You need to purchase this product separately from Oracle Corp. (see http://www.oracle.com[2]).

- **users** - they are identified by login name and password (each limited to 16 symbols). A user may be member of one or more groups.
- **groups** - identified by name (up to 128 symbols).
- **session** - each user must log into the datastore (by providing name, password and group) in order to use its resources. A session is opened when the user logs in. The default inactivity period after which the session expires and the user should log into the datastore again is 4 hours.
- **access modes** - there are four access modes in the present implementation. The access (Read/Write) to a resource according to its owner and access mode is shown in Table 9.1.

| Mode | Owner (R/W) | Owner's group (R/W) | Other users (R/W) |
|---|---|---|---|
| World Read/ Group Write | +/+ | +/+ | +/- |
| Group Read/ Group Write | +/+ | +/+ | -/- |
| Group Read/ Owner Write | +/+ | +/- | -/- |
| Owner Read/ Owner Write | +/+ | -/- | -/- |

Table 9.1: Access Modes

When GATE is configured for use with Oracle, a superuser and group are created:

- super user - ADMIN, password 'sesame'.
- administrative group - ADMINS.

The superuser is similar to the root user in Unix and has access to any resource despite its access mode.This user can also create or remove other users We recommend that you change the password of the superuser immediately after you have installed the Oracle support for GATE.

### 9.2.2 User/Group Administration

#### Running the administration tool

When GATE Oracle tables are first created with the database install scripts, they only contain the `ADMIN` user which is the only user who can create and modify users and groups.[3] We do not recommend using the `ADMIN` user to store/access LRs in GATE.

[3]This user is similar to the `root` user in Unix operating systems.

Instead, immediately after installing Oracle support for Gate datastores, some users and groups must be created by running the `UserGroupEditor` tool. Before running this tool, the URL to the Oracle database needs to be specified in gate.xml (either the user's own or the site-wide gate.xml). An example entry is:

¡DBCONFIG url="jdbc:oracle:thin:GATEUSER/gate@example.dcs.shef.ac.uk:1521:gate101" url1="jdbc:oracle:thin:GATEUSER/gate@testdb.dcs.shef.ac.uk:1521:gate02" /¿

The example entry shows that there are two databases configured for this site, one at each URL. There is no limit to the number of Oracle databases one can have, but they all need to have an attribute starting with "url", e.g., url1, url2.

To run the tool, call the gate script with the `-a` parameter.

When the tool starts up, it first asks you to select which Oracle database you wish to administer. All databases defined in the ¡DBCONFIG¿ section of gate.xml will be shown in a listbox. Once the database is chosen, a login dialog is shown, asking for the user name, password and group of the `ADMIN` user. The initial password of the `ADMIN` user is `sesame` and the group is `ADMINS`. We advise that these are changed, the first time this tool is run.

If all login credentials are provided correctly, the graphical tool starts up:

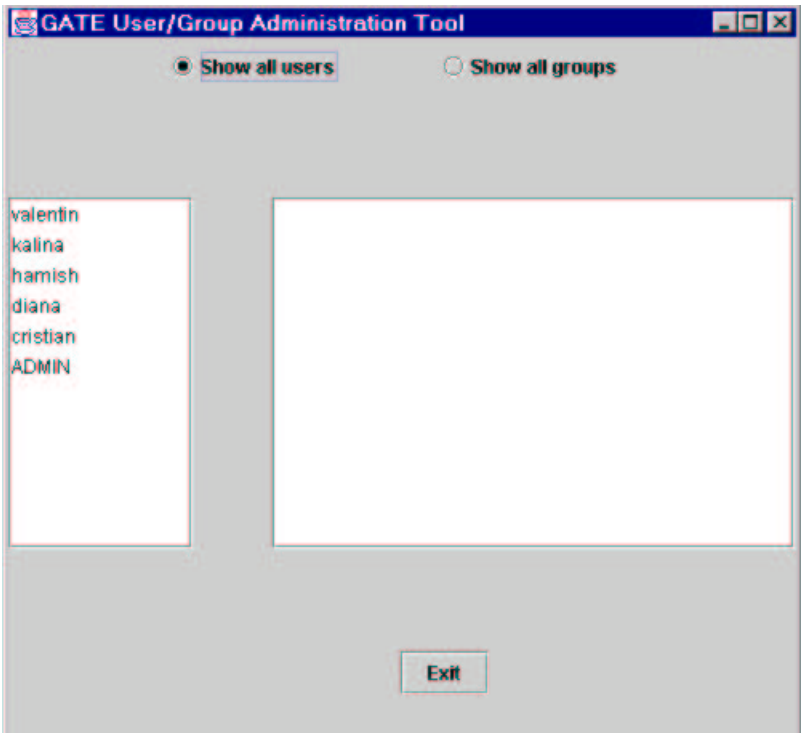



Figure 9.1: The User/Group Administration Tool

### Viewing user and group information

As shown in Figure 9.1, the user/group administration tool (called the UG tool for the rest of this section) consist of two parallel lists. By default, the left one shows a list of all users in the database and the right one is empty.

To view the groups to which a particular user belongs, you need to select that user in the list. Then the right list displays this user's groups. If the list remains empty, then it means that this user does not belong to any group.

In order to view all groups which are available, you need to switch the tool to a `Users for groups` mode, by clicking on the corresponding radio button. This will switch the tool to showing the list of all groups in the left panel. When you select a given group, then the right panel shows all users who belong to that group (see Figure 9.2).

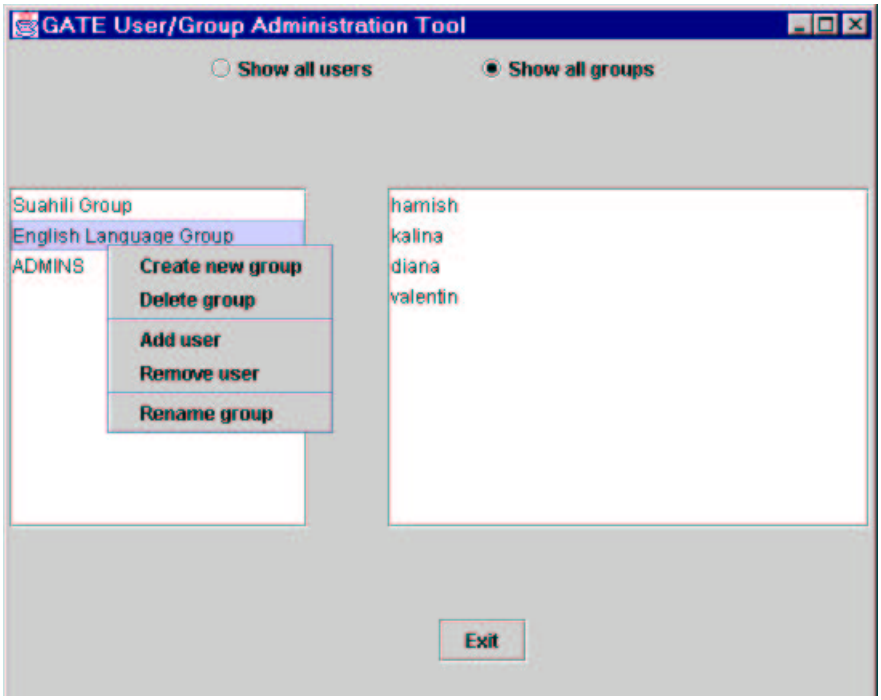



Figure 9.2: The tool in a group administration mode

### User manipulation

Users are manipulated by selecting a user in the list of users and right-clicking on it to see the user manipulation menu. This menu allows the following actions:

**Create new user:** shows a dialog where the user name and password of the new user must be specified.

**Delete user:** delete the currently selected user.

**Add to group:** shows a dialog displaying all available groups. Select one to add the user to it.

**Remove from group:** in the given dialog, choose the group from which the user is to be removed.

**Change password:** shows a dialog where the new password can be specified;

**Rename user:** choose another name for the selected user.

All changes are automatically written to the Oracle database.

#### Group manipulation

Groups are manipulated by selecting a group in the list of groups and right-clicking on it to see the group manipulation menu. This menu allows the following actions:

**Create new group:** shows a dialog where the name of the new group must be specified.

**Delete group:** delete the currently selected group.

**Add user:** shows a dialog displaying all available users. Select one to add to the group.

**Remove user:** in the given dialog, choose the user to be removed.

**Rename group:** choose another name for the selected group.

All changes are automatically written to the Oracle database.

### 9.2.3 The API

In order to work with users and groups[4] programmatically, you need to use an access controller, which is the class that provides the connection to the Oracle database. The access controller needs to be closed before application exit.

Once the connection is established, you need to create a session by proving the login details of the user (user name, password and group). Any user who can login, can use the

[4]See the latest API documentation online at: http://gate.ac.uk/gate/doc/javadoc/index.html[5]. User and group API is located in the `gate.security` package.

accessor methods for users/groups, but only the `ADMIN` user has priviliges to modify the data. The way to check whether the logged in user has the right to modify data, is to use the `isPriviligedSession()` method (see below). If a mutator method is used with a non-priviliged session, a `SecurityException` is thrown. All security-related classes and all their methods are documented in the Gate JavaDoc documentation, `java.security` package.

```
AccessController ac = new AccessControllerImpl();
ac.open("jdbc:oracle:thin:GATEUSER/gate@machine.ac.uk:1521:GateDB");

Session mySession = null;
try {
  mySession = ac.login("myUser", "myPass",ac.findGroup("myGroup").getID());
} catch (gate.security.SecurityException ex) {
  ac.close();
  <print some error and exit>
}

//first check whether we have a valid session
if (! ac.isValidSession(mySession)){
  ac.close();
  <print some error and exit>
}

//then check that it is an administrative session
if (!mySession.isPrivilegedSession()) {
  ac.close();
  <print some error and exit>
}

User myUser = ac.findUser("myUser");
String myName = myUser.getName()
List myGroups = myUser.getGroups();
...
<more code to access/modify groups and users here>

//we're done now, just close the access controller connection
ac.close();
```

If you'd like to use a dialog, where the user can type those details, the session can be obtained by using the `login(AccessController ac, Component parent)` static method in the `UserGroupEditor` class. The login code would then look as follows:

```
mySession = UserGroupDialog.login(ac, someParentWindow);
```

For a full example of code using the security API, see `TestSecurity.java` and

`UserGroupEditor.java`.

# Appendices

# Appendix A

# Design Notes

> Why has the pleasure of slowness disappeared? Ah, where have they gone, the amblers of yesteryear? Where have they gone, those loafing heroes of folk song, those vagabonds who roam from one mill to another and bed down under the stars? Have they vanished along with footpaths, with grasslands and clearings, with nature? There is a Czech proverb that describes their easy indolence by a metaphor: 'they are gazing at God's windows.' A person gazing at God's windows is not bored; he is happy. In our world, indolence has turned into having nothing to do, which is a completely different thing: a person with nothing to do is frustrated, bored, is constantly searching for an activity he lacks.
>
> *Slowness*, Milan Kundera, 1995 (pp. 4-5).

GATE is a backplane into which specialised Java Beans plug. These beans are loose-coupled with respect to each other - they communicate entirely by means of the GATE framework. Inter-component communication is handled by model components - LanguageResources, and events.

Components are defined by conformance to various interfaces (e.g. LanguageResource), ensuring separation of interface and implementation.

The reason for adding to the normal bean initialisation mech is that LRs, PRs and VRs all have characteristic parameterisation phases; the GATE resources/components model makes explicit these phases.

## A.1 Patterns

GATE is structured around a number of what we might call principles, or patterns, or alternatively, clever ideas stolen from better minds than mine. These patterns are:

- modelling most things as extensible sets of components (cf. Section A.1.1);
- separating components into model, view, or controller (cf. Section A.1.2) types;
- hiding implementation behind interfaces (cf. Section A.1.3).

Four interfaces in the top-level package describe the GATE view of components: Resource, ProcessingResource, LanguageResource and VisualResource.

### A.1.1 Components

#### Architectural Principle

Wherever users of the architecture may wish to extend the set of a particular type of entity, those types should be expressed as components.

Another way to express this is to say that the architecture is based on *agents*. I've avoided this in the past because of an association between this term and the idea of bits of code moving around between machines of their own volition. I take this to be somewhat pointless, and probably the result of an anthropomorphic obsession with mobility as a correlate of intelligence. If we drop this connotation, however, we can say that GATE is an agent-based architecture. If we want to, that is.

#### Framework Expression

Many of the classes in the framework are components, by which we mean classes that conform to an interface with certain standard properties. In our case these properties are based on the Java Beans component architecture, with the addition of component metadata, automated loading and standardised storage, threading and distribution.

All components inherit from Resource, via one of:

- LanguageResource (LR) represents entities such as lexicons, corpora or ontologies;
- VisualResource (VR) represents visualisation and editing components that participate in GUIs;
- ProcessingResource (PR) represents entities that are primarily algorithmic, such as parsers, generators or ngram modellers.

### A.1.2 Model, view, controller

According to Buschmann et al (Pattern-Oriented Software Architecture, 1996), the Model-View-Controller (MVC) pattern

> ...divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model. [p.125]

A variant of MVC, the Document-View pattern,

> ...relaxes the separation of view and controller... The View component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system.

A benefit of both arrangements is that

> ...loose coupling of the document and view components enables multiple simultaneous synchronized but different views of the same document.

Geary (Graphic Java 2, 3rd Edtn., 1999) gives a slightly different view:

> MVC separates applications into three types of objects:
>
> - **Models:** Maintain data and provide data accessor methods
> - **Views:** Paint a visual representation of some or all of a model's data
> - **Controllers:** Handle events ... By encapsulating what other architectures intertwine, MVC applications are much more flexible and reusable than their traditional counterparts.
>
> [pp. 71, 75]

Swing, the Java user interface framework, uses

> a specialised version of the classic MVC meant to support pluggable look and feel instead of applications in general. [p. 75]

GATE may be regarded as an MVC architecture in two ways:

- directly, because we use the Swing toolkit for the GUIs;
- by analogy, where LRs are models, VRs are views and PRs are controllers. Of these, the latter sits least easily with the MVC scheme, as PRs may indeed be controllers but may also not be.

### A.1.3 Interfaces

#### Architectural Principle

The implementation of types should generally be hidden from the clients of the architecture.

#### Framework Expression

With a few exceptions (such as for utility classes), clients of the framework work with the `gate.*` package. This package is mostly composed of interface definitions. Instantiations of these interfaces are obtained via the `Factory` class.

The subsidiary packages of GATE provide the implementations of the `gate.*` interfaces that are accessed via the factory. They themselves avoid directly constructing classes from other packages (with a few exceptions, such as JAPE's need for unattached annotation sets). Instead they use the factory.

## A.2 Exception Handling

When and how to use exceptions? Borrowing from Bill Venners, here are some **guidelines** (with examples):

1. Exceptions exist to refer problem conditions up the call stack to a level at which they may be dealt with. "If your method encounters an abnormal condition *that it can't handle*, it should throw an exception." If the method can handle the problem rationally, it should catch the exception and deal with it.

   **Example:**
   If the creation of a resource such as a document requires a URL as a parameter, the method that does the creation needs to construct the URL and read from it. If there is an exception during this process, the GATE method should abort by throwing its own exception. The exception will be dealt with higher up the food chain, e.g. by asking the user to input another URL, or by aborting a batch script.

2. All GATE exceptions should inherit from gate.util.GateException (a descendant of java.lang.Exception, hence a checked exception) or gate.util.GateRuntimeException (a descendant of java.lang.RuntimeException, hence an unchecked exception). This rule means that clients of GATE code can catch all sorts of exceptions thrown by the system with only two catch statements. (This rule may be broken by methods that are not public, so long as their callers catch the non-GATE exceptions and deal with them or convert them to GateException/GateRuntimeException.) Almost **all** exceptions thrown by GATE should be checked exceptions: the point of an exception is that clients of your code get to know about it, so use a checked exception to make the compiler force them to deal with it. Except:

   **Example:**
   With reference to the previous example, a problem using the URL will be signalled by something like an UnknownHostException or an IOException. These should be caught and re-thrown as descendants of GateException.

3. In a situation where an exceptional condition is an indication of a bug in the GATE library, or in the implementation of some other library, then it is permissible to throw an unchecked exception.

   **Example:**
   If a method is creating annotations on a document, and before creating the annotations it checks that their start and end points are valid ranges in relation to the content of the document (i.e. they fall within the offset space of the document, and the end is after the start), then if the method receives an InvalidOffsetException from the AnnotationSet.add call, something is seriously wrong. In such cases it may be best to throw a GateRuntimeException.

4. Where you are inheriting from a non-GATE class and therefore have the exception signatures fixed for you, you may add a new exception deriving from a non-GATE class.

   **Example:**
   The SAX XML parser API uses SaxException. Implementing a SAX parser for a document type involves overiding methods that throw this exception. Where you want to have a subtype for some problem which is specific to GATE processing, you could use GateSaxException which extends SaxException.

5. Test code is different: in the JUnit test cases it is fine just to declare that each method throws Exception and leave it at that. The JUnit test runner will pick up the exceptions and report them to you. Test methods should, however, try and ensure that the exceptions thrown are meaningful. For example, avoid null pointer exceptions in the test code itself, e.g. by using assertNonNull.

**Example:**

```
public void testComments() throws Exception {
  ResourceData docRd = (ResourceData) reg.get("gate.Document");
  assertNotNull("testComments: couldn't find document res data", docRd);
  String comment = docRd.getComment();
  assert(
    "testComments: incorrect or missing COMMENT on document",
    comment != null && comment.equals("GATE document")
  );
} // testComments()
```

See also the testing notes.

6. "Throw a different exception type for each abnormal condition." You can go too far on this one - a hundred exception types per package would certainly be too much - but in general you should create a new exception type for each different sort of problem you encounter.

   **Example:**
   The gate.creole package has a ResourceInstantiationException - this deals with all problems to do with creating resources. We could have had "ResourceUrlProblem" and "ResourceParameterProblem" but that would probably have ended up with too many. On the other hand, just throwing everything as GateException is too coarse (Hamish take note!).

7. Put exceptions in the package that they're thrown from (unless they're used in many packages, in which case they can go in gate.util). This makes it easier to find them in the documentation and prevents name clashes.

   **Example:**
   gate.jape.ParserException is correctly placed; if it was in gate.util it might clash with, for example, gate.xml.ParserException if there was such.

# Appendix B

# JAPE: Implementation

> The annual Diagram prize for the oddest book title of the year has been awarded to Gerard Forlin's Butterworths Corporate Manslaughter Service, a hefty law tome providing guidance and analysis on corporate liability for deaths in the workplace.
>
> The book, not published until January, was up against five other shortlisted titles: Fancy Coffins to Make Yourself; The Flat-Footed Flies of Europe; Lightweight Sandwich Construction; Tea Bag Folding; and The Art and Craft of Pounding Flowers: No Paint, No Ink, Just a Hammer! The shortlist was thrown open to readers of the literary trade magazine The Bookseller, who chose the winner by voting on the magazine's website. Butterworths Corporate Manslaughter Service, a snip at 375, emerged as the overall victor with 35
>
> The Diagram prize has been a regular on the award circuit since 1978, when Proceedings of the Second International Workshop on Nude Mice carried off the inaugural award. Since then, titles such as American Bottom Archaeology and last year's winner, High-Performance Stiffened Structures (an engineering publication), have received unwonted publicity through the prize. This year's winner is perhaps most notable for its lack of entendre.
>
> *Manslaughter Service kills off competition in battle of strange titles*, Emma Yates, The Guardian, November 30, 2001.

This chapter gives implementation details and formal definitions of the JAPE annotation patterns language. Section B.1 gives a more formal definition of the JAPE grammar, and some examples of its use. Section B.2 describes JAPE's relation to CPSL. The next 3 sections describe the algorithms used, label binding, and the classes used. Section B.6 gives an example of the implementation; and finally, section B.7 explains the compilation process.

## B.1 Formal Description of the JAPE Grammar

JAPE is similar to CPSL (a Common Pattern Specification Language, developed in the TIPSTER programme by Doug Appelt and others), with a few exceptions. Figure B.1 gives a BNF (Backus-Naur Format) description of the grammar.

An example rule LHS:

```
Rule: KiloAmount
( ({Token.kind == "containsDigitAndComma"}):number
  {Token.string == "kilograms"} ):whole
```

A basic constraint specification appears between curly braces, and gives a conjunction of annotation/attribute/value specifiers which have to match at a particular point in the annotation graph. A complex constraint specification appears within round brackets, and may be bound to a label with the ":" operator; the label then becomes available in the RHS for access to the annotations matched by the complex constraint. Complex constraints can also have Kleene operators (*, +, ?) applied to them. A sequence of constraints represents a sequential conjunction; disjunction is represented by separating constraints with "`|`".

Converted to the format accepted by the JavaCC LL parser generator, the most significant fragment of the CPSL grammar (as described by Appelt, based on an original specification from a TIPSTER working group chaired by Boyan Onyshkevych) goes like this:

```
constraintGroup -->
    (patternElement)+ ("|" (patternElement)+ )*

patternElement -->
    "{" constraint ("," constraint)* "}"
|   "(" constraintGroup ")" (kleeneOp)? (binding)?
```

Here the first line of `patternElement` is a basic constraint, the second a complex one.

```
MultiPhaseTransducer ::=
   ( <multiphase> <ident> )?
   ( ( SinglePhaseTransducer )+ | ( <phases> ( <ident> )+ ) )
   <EOF>
SinglePhaseTransducer ::=
   <phase> <ident> ( <input> ( <ident> )* )?
   ( <option> ( <ident> <assign> <ident> )* )?
   ( ( Rule ) | MacroDef )*
Rule ::=
   <rule> <ident> ( <priority> <integer> )?
   LeftHandSide "-->" RightHandSide
MacroDef ::=
   <macro> <ident> ( PatternElement | Action )
LeftHandSide ::=
   ConstraintGroup
ConstraintGroup ::=
   ( PatternElement )+ ( <bar> ( PatternElement )+ )*
PatternElement ::=
   ( <ident> | BasicPatternElement | ComplexPatternElement )
BasicPatternElement ::=
   ( ( <leftBrace> Constraint ( <comma> Constraint )* <rightBrace> )
     | ( <string> ) )
ComplexPatternElement ::=
   <leftBracket> ConstraintGroup <rightBracket>
   ( <kleeneOp> )? ( <colon> ( <ident> | <integer> ) )?
Constraint ::=
   ( <pling> )? <ident> ( <period> <ident> <equals> AttrVal )?
AttrVal ::=
   ( <string> | <ident> | <integer> | <floatingPoint> | <bool> )
RightHandSide ::=
   Action ( <comma> Action )*
Action ::=
   ( NamedJavaBlock | AnonymousJavaBlock | AssignmentExpression | <ident> )
NamedJavaBlock ::=
   <colon> <ident> <leftBrace> ConsumeBlock
AnonymousJavaBlock ::=
   <leftBrace> ConsumeBlock
AssignmentExpression ::=
   ( <colon> | <colonplus> ) <ident> <period> <ident>
   <assign> <leftBrace> (
     <ident> <assign>
     ( AttrVal | ( <colon> <ident> <period> <ident> <period> <ident> ) )
     ( <comma> )?
   )* <rightBrace>
ConsumeBlock ::=
   Java code
```

Figure B.1: BNF of JAPE's grammar

An example of a complete rule:

```
Rule: NumbersAndUnit
( ( {Token.kind == "number"} )+:numbers {Token.kind == "unit"} )
-->
:numbers.Name = { rule = "NumbersAndUnit" }
```

This says 'match sequences of numbers followed by a unit; create a Name annotation across the span of the numbers, and attribute rule with value NumbersAndUnit'.

## B.2 Relation to CPSL

We *differ from the CPSL spec* in various ways:

1. No pre- or post-fix context is allowed on the LHS.
2. No function calls on the LHS.
3. No string shorthand on the LHS.
4. We have two rule application algorithms (one like TextPro, one like Brill/Mitre). See section B.3.
5. Expressions relating to labels unbound on the LHS are not evaluated on the RHS. (In TextPro they evaluate to "false".) See the binding scheme description in section B.4.
6. JAPE allows arbitrary Java code on the RHS.
7. JAPE has a different macro syntax, and allows macros for both the RHS and LHS.
8. JAPE grammars are compiled and stored as serialised Java objects.

Apart from this, it is a full implementation of CPSL, and the formal power of the languages is the same (except that a JAPE RHS can delete annotations, which straight CPSL cannot). The rule LHS is a regular language over annotations; the rule RHS can perform arbitrary transformations on annotations, but the RHS is only fired *after* the LHS been evaluated, and the effects of a rule application can only be referenced after the phase in which it occurs, so the recognition power is no more than regular.

## B.3 Algorithms for JAPE Rule Application

JAPE rules are applied in one of two ways: Brill-style, where each rule is applied at every point in the document at which it matches; Appelt-style, where only the longest matching rule is applied at any point where more than one might apply.

In the Appelt case, the rule set for a phase may be considered as a single disjunctive expression (and an efficient implementation would construct a single automaton to recognise the whole rule set). To solve this problem, we need to employ two algorithms:

- one that takes as input a CPSL representation and builds a machine capable of recognizing the situations that match the rules and makes the bindings that occur each time a rule is applied. This machine is a Finite State Machine (FSM), somewhat similar to a lexical analyser (a deterministic finite state automaton).
- another one that uses the FSM built by the above algorithm and traverses the annotation graph in order to find the situations that the FSM can recognise.

### B.3.1 The first algorithm

The first step that needs to be taken in order to create the FSM is to read the CPSL description from the external file(s). This is already done in the old version of Jape.

The second step is to build a nondeterministic FSM from the java objects resulted from the parsing process. This FSM will have one initial state and a set of final states, each of them being associated to one rule (this way we know what RHS we have to execute in case of a match). The nondeterministic FSM will also have empty transitions (arcs labeled with **nil**). In order to build this FSM we will need to implement a version of the algorithm used to convert regular expressions in NFAs.

Finally, this nondeterministic FSM will have to be converted to a deterministic one. The deterministic FSM will have more states (in the worst case s! (where s is the number of states in the nondeterministic one); this case is very improbable) but will be more efficient because it will not have to backtrack.

Let **NFSM** be the nondeterministic FSM and **DFSM** the deterministic one.

The issues that have to be addressed are:

The NFSM will basically be a big OR. This means that it will have an initial state from which empty transitions will lead to the sub-FSMs associated to each rule (see Fig. B.2). When the NFSM is converted to a DFSM the initial state will be the set containing all the initial states of the FSMs associated to each rule. From that state we will have to compute the possible transitions. For this, the classical algorithm requires us to check for each possible

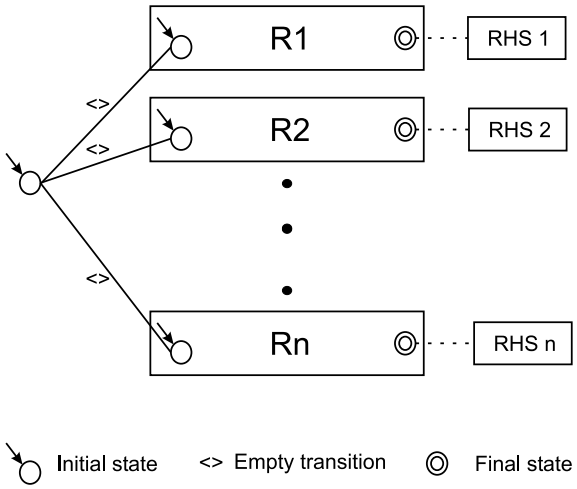



Figure B.2: A nondeterministic FSM

input symbol what is the set of reachable states. The problem is that our input symbols are actually sets of restrictions. This is similar to an automaton that has an infinite set of input symbols (although any given set of rules describes a finite set of constraints). This is not so bad, the real problem is that we have to check if there are transitions that have the same restrictions. We can safely consider that there are no two transitions with the same set of restrictions. This is safe because if this assumption is wrong, the result will be a state that has two transitions starting from it, transitions that consume the same symbol. This is not a problem because we have to check all outgoing transitions anyway; we will only check the same transition twice.

This leads to the next issue. Imagine the next part of the transition graph of a FSM (Fig. B.3):

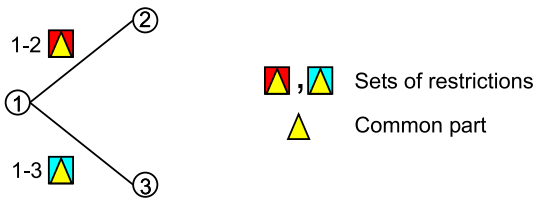



Figure B.3: Example of transitions

The restrictions associated to a transition are depicted as graphical figures (the two coloured squares). Now imagine that the two sets of restrictions have a common part (the yellow triangle).

Let us assume that at one moment the current node in the FSM graph (for one of the active FSM instances) is state 1. We get from the annotation graph the set of annotations starting from the associated current node in the annotation graph and try to advance in the FSM transition graph. In order to do this we will have to find a subset of annotations that match the restrictions for moving to state 2 or state 3. In a classical algorithm what we would do is to try to match the annotations against the restrictions "1-2" (this will return a boolean value and a set of bindings) and then we will try the matching against the restrictions "1-3" this means that we will try to match the restrictions in the common part **twice**. Because of the probable structure of the FSM transition graph there will be a lot of transitions starting from the same node which means that may be a lot of conditions checked more than one times.

What can we do to improve this?

We need a way to combine all the restrictions associated to all outgoing arcs of a state (see Fig. B.4).

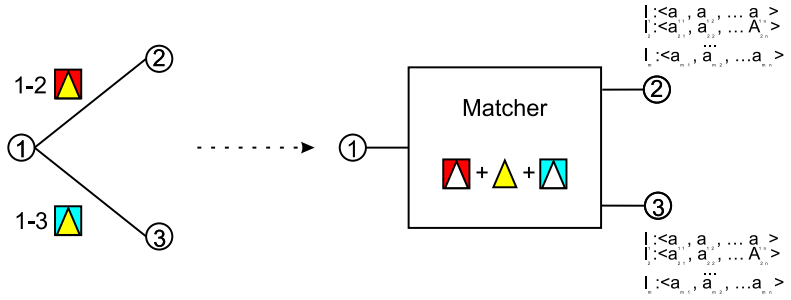



Figure B.4: A combined matching process

One way to do the (combined) matching is to pre-process the DFSM and to convert all transitions to matchers (as in Fig. B.4). This could be done using the following algorithm:

- **Input:** A DFSM;
- **Output:** A DFSM with compound restrictions checks.
- for each state s of the DFSM
    1. collect all the restrictions in the labels of the outgoings arcs from s (in the DFSM transition graph)
       **Note:** these restrictions are either of form "Type == $t_1$" or of form "Type == $t_1$ && $Attr_i == Value_i$
    2. Group all these restrictions by type and branch and create compound restrictions of form "[Type == $t_1$ && $Attr_1 == Value_1$ && $Attr_2 == Value_2$ && ... &&

$Attr_n == Value_n$]"

The grouping has to be done with care so it doesn't mix restrictions from different branches, creating unnecessary restrictive queries. These restrictions will be sent to the annotation graph which will do the matching for us. Note that we can only reuse previous queries if the restrictions are identical on two branches.[1]

3. Create the data structures necessary for linking the bindings to the results of the queries (see Fig B.5)

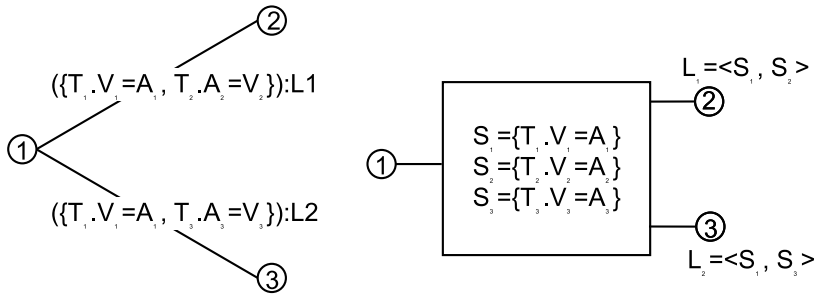



Figure B.5: Building a compound matcher

When this machine will be used for the actual matching the three queries will be run and the results will be stored in sets of annotations (S1..S3 in the picture) and...

- For each pair of annotations from ($A_1$, $A_2$) s.t. $A_1$ in $S_1$ & $A_2$ in $S_2$
  1. a new DFSM instance will be created;
  2. this instance will move to state 2;
  3. ¡$A_1$, $A_2$¿ will be bound to $L_1$
  4. the corresponding node in the annotation graph will become max($A_1$ endNode(), $A_2$.endNode()).

- Similarly, for each pair of annotations from ($A_1$, $A_3$) s.t. $A_1$ in $S_1$ & $A_3$ in $S_3$
  1. a new DFSM instance will be created;
  2. this instance will move to state 3;
  3. ¡$A_1$, $A_3$¿ will be bound to $L_2$

[1]By this we mean restrictions referring to the same type of annotations. If for branches 1-2 and 1-3 the restrictions for the type $T_1$ are the same, the query for type $T_1$ will be run only once. Each of the two branches can also have restrictions for other types of annotations.

4. the corresponding node in the annotation graph will become max($A_1$.endNode(), $A_3$.endNode()).

While building the compound matcher it is possible to detect queries that depend one from another (e.g. if the expected results of a query are a subset of the results from another query). This kind of situations can be marked so when the queries are actually run some operations can be avoided (e.g. if the less restrictive search returned no results than the more restrictive one can be skipped, or if a search returns an AnnotationSet (an object that can be queried) than the more restrictive query can be.

### B.3.2 Algorithm 2

Consider the following figure:

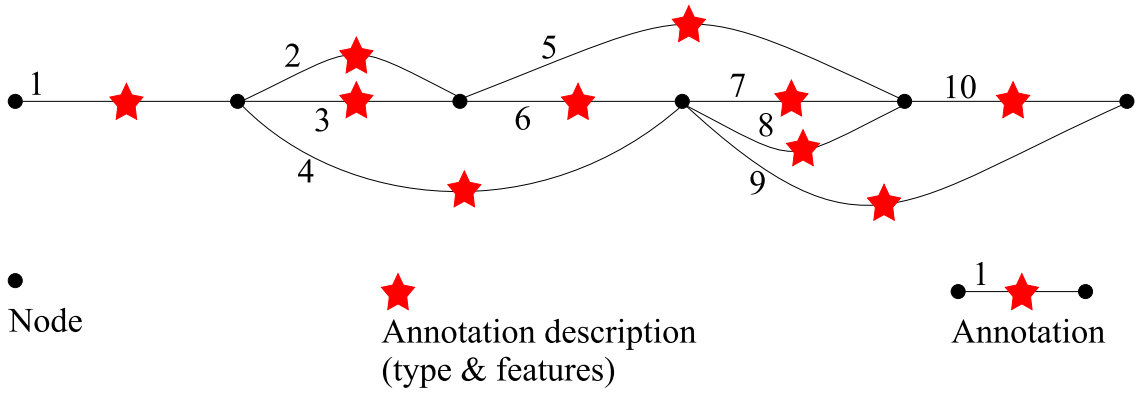



Figure B.6: An annotation graph

Basically, the algorithm has to traverse this graph starting from the leftmost node to the rightmost one. Each path found is a sequence of possible matches.

Because more than one annotation (all starting at the same point) can be matched at one step, a path is not viewed as a classical path in a graph, but a sequence of steps, each step being a set of annotations that start in the same node.
*e.g. a path in the graph above can be: [1].[2,4].[7,8].[10];*
*Note that the next step continues from the rightmost node reached by the annotations in the current step.*

The matchings are made by a Finite State Machine that resembles an clasical lexical analyser (*aka. scanner*). The main difference from a scanner is that there are no input symbols; the transition from one state to another is based on matching a set of objects (annotations) against a set of restrictions (the constraint group in the LHS of a CPSL rule).

The algorithm can be the following:

1. startNode = the leftmost node
2. create a first instance of the FSM and add it to the list of active instances;
3. for this FSM instance set current node as the leftmost node;
4. while(startNode != last node) do

    1 while (not over) do
        1 for each $Fi$ active instance of the FSM do
            1 if this instance is in a final state then save a clone of it in the set of accepting FSMs (instances of the FSM that have reached a final state);
            2 read all the annotations starting from the current node;
            3 select all sets of annotation that can be used to advance one step in the transition graph of the FSM;
            4 for each such set create a new instance of the FSM, put it in the active list and make it consume the corresponding set of annotations, making any necessary bindings in the process (this new instance will advance in the annotation graph to the rightmost node that is an end of a matched annotation);
            5 discard $F_i$;
        2 end for;
        3 if the set of active instances of FSM is empty * then over = true;

        end while;
    2 if the set of accepting FSMs is not empty
        1 from all accepting FSMs select ** the one that matched the longest path;if there are more than one for the same path length select the one with highest priority;
        2 execute the action associated to the final state of the selected FSM instance;
        3 startNode = selectedFSMInstance.getLastNode.getNextNode();
    3 else //the matching failed → start over from the next node // startNode = startNode.getNextNode();

5. end while;

**: the set of active FSM instances can decrease when an active instance cannot continue (there is no set of annotations starting from its current node that can be matched). In this case it will be removed from the set.*

***: if we do Brill style matching, we have to process each of the accepting instances.*

## B.4 Label Binding Scheme

In TextPro, a ":" label binds to the last matched annotation in its scope. A "+:" label binds to all the annotations matched in the scope. In JAPE there is no "+:" label (though there is a ":+" – see below), due to the ambiguity with Kleene +. In CPSL a constraint group can be both labelled and have a Kleene operator. How can Kleene + followed by label : be distinguished from label +: ? E.g. given `(....)+:label` are the constraints within the brackets having Kleene + applied to them and being labelled, or is it a +: label?

Appelt's answer is that +: is always a label; to get the other interpretation use `((...)+):`. This may be difficult for rule developers to remember; JAPE disallows the "+:" label, and makes all matched annotations available from every label.

JAPE adds a ":+" label operator, which means that all the spans of any annotations matched are assigned to new annotations created on the RHS relative to that label. (With ordinary ":" labels, only the span of the outermost corners of the annotations matched is used.) (This operator disappears in GATE version 2, with the elimination of multi-span annotations.)

Another problem regards RHS interpretation of unbound labels. If we have something like

```
(
  ( {Word.string == "thing"} ):1
  |
  ( {Word.string == "otherthing"} ):2
)
```

on the LHS, and references to :1 and :2 on the RHS, only one of these will actually be bound to anything when the rule is fired. The expression containing the other should be ignored. In TextPro, an assignment on the RHS that references an unbound label is evaluated to the value "false". In JAPE, RHS expressions involving unbound operators are not evaluated.

## B.5 Classes

The main external interfaces to JAPE are the classes `gate.jape.Batch` and `gate.jape.Compiler`. The CPSL Parser is implemented by `ParseCpsl.jj`, which is input to JavaCC (and JJDoc to produce grammar documentation) and finally Java itself. (There are lots of other classes produced along the way by the compiler-compiler tools:

`ASCII_CharStream.java JJTParseCpslState.java Node.java ParseCpsl.java ParseCpslConstants.java ParseCpslTokenManager.java ParseCpslTreeConstants.java ParseException.java SimpleNode.java TestJape.java Token.java TokenMgrError.java`

These live in the parser subpackage, in the `gate/jape/parser` directory.

Each grammar results in an object of class `Transducer`, which has a set of `Rule`.

Constants are held in the interface `JapeConstants`. The test harness is in `TestJape`.

## B.6 Implementation

### B.6.1 A Walk-Through

The pattern application algorithm (which is either like Doug's, or like Brill's), makes a top-level call to something like

```
boolean matches(int position, Document doc,
                MutableInteger newPosition)
throws PostionOutOfRange
```

which is a method on each `Rule`. This is in turn deferred to the rule's `LeftHandSide`, and thence to the `ConstraintGroup` which each `LeftHandSide` contains. The `ConstraintGroup` iterates over its set of `PatternElementConjunctions`; when one succeeds, the matches call returns true; if none succeed, it returns false. The `Rules` also have

```
void transduce(Document doc) throws LhsNotMatched
```

methods, which may be called after a successful match, and result in the application of the `RightHandSide` of the `Rule` to the document.

`PatternElements` also implement the matches method. Whenever it succeeds, the annotations which were consumed during the match are available from that element, as are a composite span set, and a single span that covers the whole set. In general these will only be accessed via a `bindingName`, which is associated with `ComplexPatternElements`. The `LeftHandSide` maintains a mapping of `bindingNames` to `ComplexPatternElements` (which are accessed by array reference in `Rule RightHandSides`).

Although `PatternElements` give access to an annotation set, these are only built when they are asked for (caching ensures that they are only built once) to avoid storing annotations against every matched element. When asked for, the construction process is an iterative traversal of the elements contained within the element being asked for the annotations. This traversal always bottoms out into `BasicPatternElements`, which are the only ones that need to store annotations all the time.

In a `RightHandSide` application, then, a call to the `LeftHandSide's` binding environment will yield a `ComplexPatternElement` representing the bound object, from which annotations and spans can be retrieved as needed.

### B.6.2 Example RHS code

Let's imagine we are writing an RHS for a rule which binds a set of annotations representing simple numbers to the label `:numbers`. We want to create a new annotation spanning all the ones matched, whose value is an Integer representing the sum of the individual numbers.

The RHS consists of a comma-separated list of blocks, which are either anonymous or labelled. (We also allow the CPSL-style shorthand notation as implemented in TextPro. This is more limiting than code, though, e.g. I don't know how you could do the summing operation below in CPSL.) Anonymous blocks will be evaluated within the same scope, which encloses that of all named blocks, and all blocks are evaluated in order, so declarations can be made in anonymous blocks and then referenced in subsequent blocks. Labelled blocks will only be evaluated when they were bound during LHS matching. The symbol `doc` is always scoped to the Document which the `Transducer` this rule belongs to is processing. For example:

```
// match a sequence of integers, and store their sum
Rule:   NumberSum

( {Token.kind == "otherNum"} )+ :numberList

-->

:numberList{
  // the running total
  int theSum = 0;

  // loop round all the annotations the LHS consumed
  for(int i = 0; i<numberListAnnots.length(); i++) {

    // get the number string for this annot
    String numberString = doc.spanStrings(numberListAnnots.nth(i));

    // parse the number string and add to running total
    try {
      theSum += Integer.parseInt(numberString);
    } catch(NumberFormatException e) {
      // ignore badly-formatted numbers
    }
  } // for each number annot

  doc.addAnnotation(
    "number",
```

```
    numberListAnnots.getLeftmostStart(),
    numberListAnnots.getRightmostEnd(),
    "sum",
    new Integer(theSum)
  );

} // :numberList
```

This stuff then gets converted into code (that is used to form the class we create for RHSs) looking like this:

```
package japeactionclasses;

import gate.*; import java.io.*; import gate.jape.*;
import gate.util.*; import gate.creole.*;

public class Test2NumberSumActionClass
implements java.io.Serializable, RhsAction {

  public void doit(Document doc, LeftHandSide lhs) {

    AnnotationSet numberListAnnots = lhs.getBoundAnnots("numberList");
    if(numberListAnnots.size() != 0) {
      int theSum = 0;

      for(int i = 0; i<numberListAnnots.length(); i++) {
        String numberString = doc.spanStrings(numberListAnnots.nth(i));

        try {
          theSum += Integer.parseInt(numberString);
        } catch(NumberFormatException e) {    }
      }

      doc.addAnnotation(
        "number",
        numberListAnnots.getLeftmostStart(),
        numberListAnnots.getRightmostEnd(),
        "sum",
        new Integer(theSum)
      );

    }
  }
}
```

## B.7 Compilation

JAPE uses a compiler that translates CPSL grammars to Java objects that target the GATE API (and a regular expression library). It uses a compiler-compiler (JavaCC) to construct the parser for CPSL. Because CPSL is a transducer based on a regular language (in effect an FST) it deploys similar techniques to those used in the lexical analysers of parser generators (e.g. lex, flex, JavaCC tokenisation rules).

In other words, the JAPE compiler is a compiler generated with the help of a compiler-compiler which uses back-end code similar to that used in compiler-compilers. Confused? If not, welcome to the domain of the nerds, which is where you belong; I'm sure you'll be happy here.

# Appendix C

# Named-Entity State Machine Patterns

> There are, it seems to me, two basic reasons why minds aren't computers... The first... is that human beings are organisms. Because of this we have all sorts of needs - for food, shelter, clothing, sex etc - and capacities - for locomotion, manipulation, articulate speech etc, and so on - to which there are no real analogies in computers. These needs and capacities underlie and interact with our mental activities. This is important, not simply because we can't understand how humans behave except in the light of these needs and capacities, but because any historical explanation of how human mental life developed can only do so by looking at how this process interacted with the evolution of these needs and capacities in successive species of hominids.
>
> ...
>
> The second reason... is that... brains don't work like computers.
>
> *Minds, Machines and Evolution*, Alex Callinicos, 1997 (ISJ 74, p.103).

This chapter describes the individual grammars used in GATE for Named Entity Recognition, and how they are combined together. It relates to the default NE grammar for ANNIE, but should also provide guidelines for those adapting or creating new grammars. For documentation about specific grammars other than this core set, use this document in combination with the comments in the relevant grammar files. chapter 5 also provides information about designing new grammar rules and tips for ensuring maximum processnig speed.

## C.1 Main.jape

This file contains a list of the grammars to be used, in the correct processing order. The ordering of the grammars is crucial, because they are processed in series, and later grammars

may depend on annotations produced by earlier grammars.

The default grammar consists of the following phases:

- first.jape
- firstname.jape
- name.jape
- name_post.jape
- date_pre.jape
- date.jape
- reldate.jape
- number.jape
- address.jape
- url.jape
- identifier.jape
- jobtitle.jape
- final.jape
- unknown.jape
- name_context.jape
- org_context.jape
- loc_context.jape
- clean.jape

## C.2 first.jape

This grammar must always be processed first. It can contain any general macros needed for the whole grammar set. This should consist of a macro defining how space and control characters are to be processed (and may consequently be different for each grammar set, depending on the text type). Because this is defined first of all, it is not necessary to restate this in later grammars. This has a big advantage – it means that default grammars can be used for specialised grammar sets, without having to be adapted to deal with e.g. different

treatment of spaces and control characters. In this way, only the first.jape file needs to be changed for each grammar set, rather than every individual grammar.

The first.jape grammar also has a dummy rule in. This is never intended to fire – it is simply added because every grammar set must contain rules, but there are no specific rules we wish to add here. Even if the rule were to match the pattern defined, it is designed not to produce any output (due to the empty RHS).

## C.3 firstname.jape

This grammar contains rules to identify first names and titles via the gazetteer lists. It adds a gender feature where appropriate from the gazeteer list. This gender feature is used later in order to improve co-reference between names and pronouns. The grammar creates separate annotations of type FirstPerson and Title.

## C.4 name.jape

This grammar contains initial rules for organization, location and person entities. These rules all create temporary annotations, some of which will be discarded later, but the majority of which will be converted into final annotations in later grammars. Rules beginning with "Not" are negative rules – this means that we detect something and give it a special annotation (or no annotation at all) in order to prevent it being recognised as a name. This is because we have no negative operator (we have "=" but not "!=").

### C.4.1 Person

We first define macros for initials, first names, surnames, and endings. We then use these to recognise combinations of first names from the previous phase, and surnames from their POS tags or case information. Persons get marked with the annotation "TempPerson". We also percolate feature information about the gender from the previous annotations if known.

### C.4.2 Location

The rules for Location are fairly straightforward, but we define them in this grammar so that any ambiguity can be resolved at the top level. Locations are often combined with other entity types, such as Organisations. This is dealt with by annotating the two entity types separately, and them combining them in a later phase. Locations are recognised mainly by

gazetter lookup, using not only lists of known places, but also key words such as mountain, lake, river, city etc. Locations are annotated as TempLocation in this phase.

### C.4.3 Organization

Organizations tend to be defined either by straight lookup from the gazetteer lists, or, for the majority, by a combination of POS or case information and key words such as "company", "bank", "Services" "Ltd." etc. Many organizations are also identified by contextual information in the later phase org_context.jape. In this phase, organizations are annotated as TempOrganization.

### C.4.4 Ambiguities

Some ambiguities are resolved immediately in this grammar, while others are left until later phases. For example, a Christian name followed by a possible Location is resolved by default to a person rather than a Location (e.g. "Ken London"). On the other hand, a Christian name followed by a possible organisation ending is resolved to an Organisation (e.g. "Alexandra Pottery"), though this is a slightly less sure rule.

### C.4.5 Contextual information

Although most of the rules involving contextual information are invoked in a much later phase, there are a few which are invoked here, such as "X joined Y" where X is annotated as a Person and Y as an Organization. This is so that both annotations types can be handled at once.

## C.5 name_post.jape

This grammar runs after the name grammar to fix some erroneous annotations that may have been created. Of course, a more elegant solution would be not to create the problem in the first instance, but this is a workaround. For example, if the surname of a Person contains certain stop words, e.g. "Mary And" then only the first name should be recognised as a Person. However, it might be that the firstname is also an Organization (and has been tagged with TempOrganization already), e.g. "U.N." If this is the case, then the annotation is left untouched, because this is correct.

## C.6 date_pre.jape

This grammar precedes the date phase, because it includes extra context to prevent dates being recognised erroneously in the middle of longer expressions. It mainly treats the case where an expression is already tagged as a Person, but could also be tagged as a date (e.g. 16th Jan).

## C.7 date.jape

This grammar contains the base rules for recognising times and dates. Given the complexity of potential patterns representing such expressions, there are a large number of rules and macros.

Although times and dates can be mutually ambiguous, we try to distinguish between them as early as possible. Dates, times and years are generally tagged separately (as TempDate, TempTime and TempYear respectively) and then recombined to form a final Date annotation in a later phase. This is because dates, times and years can be combined together in many different ways, and also because there can be much ambiguity between the three. For example, 1312 could be a time or a year, while 9-10 could be a span of time or date, or a fixed time or date.

## C.8 reldate.jape

This grammar handles relative rather than absolute date and time sequences, such as "yesterday morning", "2 hours ago", "the first 9 months of the financial year"etc. It uses mainly explicit key words such as "ago" and items from the gazetteer lists.

## C.9 number.jape

This grammar covers rules concerning money and percentages. The rules are fairly straightforward, using keywords from the gazetteer lists, and there is little ambiguity here, except for example where "Pound" can be money or weight, or where there is no explicit currency denominator.

## C.10 address.jape

Rules for Address cover ip addresses, phone and fax numbers, and postal addresses. In general, these are not highly ambiguous, and can be covered with simple pattern matching, although phone numbers can require use of contextual information. Currenly only UK formats are really handled, though handling of foreign zipcodes and phone number formats is envisaged in future. The annotations produced are of type Email, Phone etc. and are then replaced in a later phase with final Address annotations with "phone" etc. as features.

## C.11 url.jape

Rules for email addresses and Urls are in a separate grammar from the other address types, for the simple reason that SpaceTokens need to be identified for these rles to operate, whereas this is not necessary for the other Address types. For speed of processing, we place them in separate grammars so that SpaceTokens can be eliminated from the Input when they are not required.

## C.12 identifier.jape

This grammar identifies "Identifiers" which basically means any combination of numbers and letters acting as an ID, reference number etc. not recognised as any other entity type.

## C.13 jobtitle.jape

This grammar simply identifies Jobtitles from the gazetteer lists, and adds a JobTitle annotation, which is used in later phases to aid recognition of other entity types such as Person and Organization. It may then be discarded in the Clean phase if not required as a final annotation type.

## C.14 final.jape

This grammar uses the temporary annotations previously assigned in the earlier phases, and converts them into final annotations. The reason for this is that we need to be able to resolve ambiguities between different entity types, so we need to have all the different entity types handled in a single grammar somewhere. Ambiguities can be resolved using prioritisation

techniques. Also, we may need to combine previously annotated elements, such as dates and times, into a single entity.

The rules in this grammar use Java code on the RHS to remove the existing temporary annotations, and replace them with new annotations. This is because we want to retain the features associated with the temporary annotations. For example, we might need to keep track of whether a person is male or female, or whether a location is a city or country. It also enables us to keep track of which rules have been used, for debugging purposes.

For the sake of obfuscation, although this phase is called final, it is not the final phase!

## C.15 unknown.jape

This short grammar finds proper nouns not previously recognised, and gives them an Unknown annotation. This is then used by the namematcher – if an Unknown annotation can be matched with a previously categorised entity, its annotation is changed to that of the matched entity. Any remaining Unknown annotations are useful for debugging purposes, and can also be used as input for additional grammars or processing resources.

## C.16 name_context.jape

This grammar looks for Unknown annotations occurring in certain contexts which indicate they might belong to Person. This is a typical example of a grammar that would benefit from learning or automatic context generation, because useful contexts are (a) hard to find manually and may require large volumes of training data, and (b) often very domain–specific. In this core grammar, we confine the use of contexts to fairly general uses, since this grammar should not be domain–dependent.

## C.17 org_context.jape

This grammar operates on a similar principle to name_context.jape. It is slightly oriented towards business texts, so does not quite fulfil the generality criteria of the previous grammar. It does, however, provide some insight into more detailed use of contexts.¡/p¿

## C.18 loc_context.jape

This grammar also operates in a similar manner to the preceding two, using general context such as coordinated pairs of locations, and hyponymic types of information.

## C.19 clean.jape

This grammar comes last of all, and simply aims to clean up (remove) some of the temporary annotations that may not have been deleted along the way.

# References


[Appelt 99]
D. Appelt. An Introduction to Information Extraction. *Artificial Intelligence Communications*, 12(3):161–172, 1999.

[Azar 89]
S. Azar. *Understanding and Using English Grammar*. Prentice Hall Regents, 1989.

[Bird & Liberman 99]
S. Bird and M. Liberman. A Formal Framework for Linguistic Annotation. Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania, 1999. `http://xxx.lanl.gov/abs/cs.CL/9903003`.

[Bontcheva *et al.* 00]
K. Bontcheva, H. Brugman, A. Russel, P. Wittenburg, and H. Cunningham. An Experiment in Unifying Audio-Visual and Textual Infrastructures for Language Processing R&D. In *Proceedings of the Workshop on Using Toolsets and Architectures To Build NLP Systems at COLING-2000*, Luxembourg, 2000. `http://gate.ac.uk/`.

[Booch 94]
G. Booch. *Object-Oriented Analysis and Design 2nd Edn.* Benjamin/Cummings, 1994.

[Brugman *et al.* 99]
H. Brugman, K. Bontcheva, P. Wittenburg, and H. Cunningham. Integrating Multimedia and Textual Software Architectures for Language Technology. Technical report MPI-TG-99-1, Max-Planck Institute for Psycholinguistics, Nijmegen, Netherlands, 1999.

[Campione *et al.* 98]
M. Campione, K. Walrath, A. Huml, and the Tutuorial Team. *The Java Tutorial Continued: The Rest of the JDK*. Addison-Wesley, Reading, MA, 1998.

[Chinchor 92]
N. Chinchor. Muc-4 evaluation metrics. In *Proceedings of the Fourth Message Understanding Conference*, pages 22–29, 1992.

[Cobuild 99]
C. Cobuild, editor. *English Grammar*. Harper Collins, 1999.

[Cowie & Lehnert 96]
J. Cowie and W. Lehnert. Information Extraction. *Communications of the ACM*, 39(1):80–91, 1996.

[Cunningham 94]
H. Cunningham. Support Software for Language Engineering Research. Technical Report 94/05, Centre for Computational Linguistics, UMIST, Manchester, 1994.

[Cunningham 99a]
H. Cunningham. A Definition and Short History of Language Engineering. *Journal of Natural Language Engineering*, 5(1):1–16, 1999.

[Cunningham 99b]
H. Cunningham. Information Extraction: a User Guide (revised version). Research Memorandum CS–99–07, Department of Computer Science, University of Sheffield, May 1999.

[Cunningham 99c]
H. Cunningham. JAPE: a Java Annotation Patterns Engine. Research Memorandum CS–99–06, Department of Computer Science, University of Sheffield, May 1999.

[Cunningham 00]
H. Cunningham. *Software Architecture for Language Engineering*. Unpublished PhD thesis, University of Sheffield, 2000. `http://gate.ac.uk/sale/thesis/`.

[Cunningham 02]
H. Cunningham. GATE, a General Architecture for Text Engineering. *[in press]*, ??(??):??, 2002. Accepted for publication by Computing and the Humanities, May 2001.

[Cunningham *et al.* 94]
H. Cunningham, M. Freeman, and W. Black. Software Reuse, Object-Oriented Frameworks and Natural Language Processing. In *New Methods in Language Processing (NeMLaP-1), September 1994*, Manchester, 1994. (Re-published in book form 1997 by UCL Press).

[Cunningham *et al.* 95]
H. Cunningham, R. Gaizauskas, and Y. Wilks. A General Architecture for Text Engineering (GATE) – a new approach to Language Engineering R&D. Technical Report CS–95–21, Department of Computer Science, University of Sheffield, 1995. `http://xxx.lanl.gov/abs/cs.CL/9601009`.

[Cunningham *et al.* 96a]
H. Cunningham, K. Humphreys, R. Gaizauskas, and M. Stower. CREOLE Developer's Manual. Technical report, Department of Computer Science, University of Sheffield, 1996. `http://www.dcs.shef.ac.uk/nlp/gate`.

[Cunningham *et al.* 96b]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. TIPSTER-Compatible Projects at Sheffield. In *Advances in Text Processing, TIPSTER Program Phase II*. DARPA, Morgan Kaufmann, California, 1996.

[Cunningham *et al.* 96c]
H. Cunningham, Y. Wilks, and R. Gaizauskas. GATE – a General Architecture for Text Engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING-96)*, Copenhagen, August 1996.

[Cunningham *et al.* 96d]
H. Cunningham, Y. Wilks, and R. Gaizauskas. Software Infrastructure for Language Engineering. In *Proceedings of the AISB Workshop on Language Engineering for Document Analysis and Recognition*, Brighton, U.K., April 1996.

[Cunningham *et al.* 96e]
H. Cunningham, Y. Wilks, and R. Gaizauskas. New Methods, Current Trends and Software Infrastructure for NLP. In *Proceedings of the Conference on New Methods in Natural Language Processing (NeMLaP-2)*, Bilkent University, Turkey, September 1996. `http://xxx.lanl.gov/abs/cs.CL/9607025`.

[Cunningham *et al.* 97a]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. GATE – a TIPSTER-based General Architecture for Text Engineering. In *Proceedings of the TIPSTER Text Program (Phase III) 6 Month Workshop*. DARPA, Morgan Kaufmann, California, May 1997.

[Cunningham *et al.* 97b]
H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks. Software Infrastructure for Natural Language Processing. In *Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP-97)*, March 1997. `http://xxx.lanl.gov/-abs/cs.CL/9702005`.

[Cunningham *et al.* 98a]
H. Cunningham, W. Peters, C. McCauley, K. Bontcheva, and Y. Wilks. A Level Playing Field for Language Resource Evaluation. In *Workshop on Distributing and Accessing Lexical Resources at Conference on Language Resources Evaluation, Granada, Spain*, 1998.

[Cunningham *et al.* 98b]
H. Cunningham, M. Stevenson, and Y. Wilks. Implementing a Sense Tagger within a General Architecture for Language Engineering. In *Proceedings of the Third Conference on New Methods in Language Engineering (NeMLaP-3)*, pages 59–72, Sydney, Australia, 1998.

[Cunningham *et al.* 99]
H. Cunningham, R. Gaizauskas, K. Humphreys, and Y. Wilks. Experience with a

Language Engineering Architecture: Three Years of GATE. In *Proceedings of the AISB'99 Workshop on Reference Architectures and Data Standards for NLP*, Edinburgh, April 1999. The Society for the Study of Artificial Intelligence and Simulation of Behaviour.

[Cunningham *et al.* 00a]
H. Cunningham, K. Bontcheva, W. Peters, and Y. Wilks. Uniform language resource access and distribution in the context of a General Architecture for Text Engineering (GATE). In *Proceedings of the Workshop on Ontologies and Language Resources (OntoLex'2000)*, Sozopol, Bulgaria, September 2000. `http://gate.ac.uk/sale/ontolex/ontolex.ps`.

[Cunningham *et al.* 00b]
H. Cunningham, K. Bontcheva, V. Tablan, and Y. Wilks. Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC-2)*, Athens, 2000. `http://gate.ac.uk/`.

[Cunningham *et al.* 00c]
H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, and Y. Wilks. Experience of using GATE for NLP R&D. In *Proceedings of the Workshop on Using Toolsets and Architectures To Build NLP Systems at COLING-2000*, Luxembourg, 2000. `http://gate.ac.uk/`.

[Cunningham *et al.* 00d]
H. Cunningham, D. Maynard, and V. Tablan. JAPE: a Java Annotation Patterns Engine (Second Edition). Research Memorandum CS–00–10, Department of Computer Science, University of Sheffield, November 2000.

[Frakes & Baeza-Yates 92]
W. Frakes and R. Baeza-Yates, editors. *Information retrieval, data structures and algorithms*. Prentice Hall, New York, Englewood Cliffs, N.J., 1992.

[Gaizauskas & Wilks 98]
R. Gaizauskas and Y. Wilks. Information Extraction: Beyond Document Retrieval. *Journal of Documentation*, 54(1):70–105, 1998.

[Gaizauskas *et al.* 96a]
R. Gaizauskas, P. Rodgers, H. Cunningham, and K. Humphreys. GATE User Guide. `http://www.dcs.shef.ac.uk/nlp/gate`, 1996.

[Gaizauskas *et al.* 96b]
R. Gaizauskas, H. Cunningham, Y. Wilks, P. Rodgers, and K. Humphreys. GATE – an Environment to Support Research and Development in Natural Language Engineering. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-96)*, Toulouse, France, October 1996.

[Gambäck & Olsson 00]
B. Gambäck and F. Olsson. Experiences of Language Engineering Algorithm Reuse. In *Second International Conference on Language Resources and Evaluation (LREC)*, pages 155–160, Athens, Greece, 2000.

[Gazdar & Mellish 89]
G. Gazdar and C. Mellish. *Natural Language Processing in Prolog.* Addison-Wesley, Reading, MA, 1989.

[Grishman 97]
R. Grishman. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA, 1997. `http://www.itl.nist.gov/div894/894.02/related_projects/-tipster/`.

[Hepple 00a]
M. Hepple. Independence and Commitment: Assumptions for Rapid Training and Execution of Rule-based Part-of-Speech Taggers. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics*, Hong Kong, October 2000.

[Hepple 00b]
M. Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000)*, Hong Kong, October 2000.

[Humphreys *et al.* 96]
K. Humphreys, R. Gaizauskas, H. Cunningham, and S. Azzam. CREOLE Module Specifications. `http://www.dcs.shef.ac.uk/nlp/gate/`, 1996.

[Jackson 75]
M. Jackson. *Principles of Program Design.* Academic Press, London, 1975.

[LREC-1 98]
*Conference on Language Resources Evaluation (LREC-1)*, Granada, Spain, 1998.

[LREC-2 00]
*Second Conference on Language Resources Evaluation (LREC-2)*, Athens, 2000.

[Manning & Schütze 99]
C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* MIT press, Cambridge, MA, 1999. Supporting materials available at `http://www.sultry.arts.usyd.edu.au/fsnlp/` .

[Maynard *et al.* 00]
D. Maynard, H. Cunningham, K. Bontcheva, R. Catizone, G. Demetriou, R. Gaizauskas, O. Hamza, M. Hepple, P. Herring, B. Mitchell, M. Oakes, W. Peters, A. Setzer, M. Stevenson, V. Tablan, C. Ursu, and Y. Wilks. A Survey of Uses of

GATE. Technical Report CS–00–06, Department of Computer Science, University of Sheffield, 2000.

[Maynard *et al.* 01]
D. Maynard, V. Tablan, C. Ursu, H. Cunningham, and Y. Wilks. Named Entity Recognition from Diverse Text Types. In *Recent Advances in Natural Language Processing 2001 Conference*, Tzigov Chark, Bulgaria, 2001.

[Maynard *et al.* 02]
D. Maynard, V. Tablan, H. Cunningham, C. Ursu, H. Saggion, K. Bontcheva, and Y. Wilks. Architectural elements of language engineering robustness. *Journal of Natural Language Engineering – Special Iisue on Robust Methods in Analysis of Natural Language Data*, 2002. forthcoming.

[McEnery *et al.* 00]
A. McEnery, P. Baker, R. Gaizauskas, and H. Cunningham. EMILLE: Building a Corpus of South Asian Languages. *Vivek, A Quarterly in Artificial Intelligence*, 13(3):23–32, 2000.

[Peters *et al.* 98]
W. Peters, H. Cunningham, C. McCauley, K. Bontcheva, and Y. Wilks. Uniform Language Resource Access and Distribution. In *Workshop on Distributing and Accessing Lexical Resources at Conference on Language Resources Evaluation*, Granada, Spain, 1998.

[Shaw & Garlan 96]
M. Shaw and D. Garlan. *Software Architecture*. Prentice Hall, New York, 1996.

[Stevenson *et al.* 98]
M. Stevenson, H. Cunningham, and Y. Wilks. Sense tagging and language engineering. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, pages 185–189, Brighton, U.K., 1998.

[The Unicode Consortium 96]
The Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison-Wesley, Reading, MA, 1996.

[van Rijsbergen 79]
C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[Yourdon 89]
E. Yourdon. *Modern Structured Analysis*. Prentice Hall, New York, 1989.

[Yourdon 96]
E. Yourdon. *The Rise and Resurrection of the American Programmer*. Prentice Hall, New York, 1996.

# Colophon

> Formal semantics (henceforth FS), at least as it relates to computational language understanding, is in one way rather like connectionism, though without the crucial prop Sejnowski's work (1986) is widely believed to give to the latter: both are old doctrines returned, like the Bourbons, having learned nothing and forgotten nothing. But FS has nothing to show as a showpiece of success after all the intellectual groaning and effort.
>
> *On Keeping Logic in its Place* (in Theoretical Issues in Natural Language Processing, ed. Wilks), Yorick Wilks, 1989 (p.130).

We wanted to be modern, we wanted to make the XML people feel like progress is indeed happening, we wanted to update our CVs with the latest trick.... So we looked into using XML as source for this document, and using something like DocBook[1] to translate it into the PDF and HTML versions that we wanted to provide for printing and web viewing. Nice ideas, but our conclusion was that they're not really ready right now. So in the end it was good old LaTeX and TeX4HT[2] for the HTML production. Thank you Don Knuth, Leslie Lamport and Eitan Gurari.

[1] `http://www.docbook.org`

[2] `http://www.cis.ohio-state.edu/~gurari/TeX4ht/mn.html`