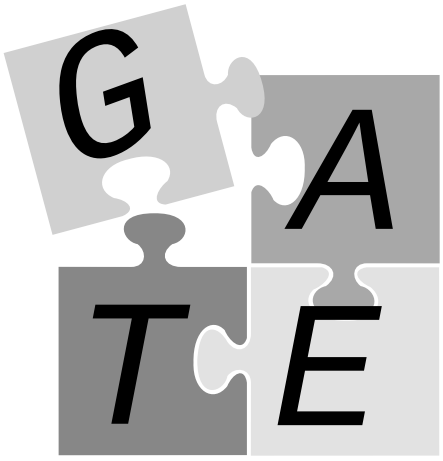
CS - 00 - 09

LaSIE Technical Specifications

K. Humphreys, R. Gaizauskas &
H. Cunningham

LaSIE Technical Specifications



Kevin Humphreys
Rob Gaizauskas
Hamish Cunningham

Department of Computer Science and
Institute for Language, Speech and Hearing (ILASH)
University of Sheffield, UK

Contact us at: gate@dcs.shef.ac.uk
Bugs to: gate-crashers@dcs.shef.ac.uk
Discussion list: gate-discuss@dcs.shef.ac.uk (admin: majordomo@dcs.shef.ac.uk)
Announcements list: gate-announce@dcs.shef.ac.uk (admin:
majordomo@dcs.shef.ac.uk)
http://www.dcs.shef.ac.uk/nlp/gate or http://gate.ac.uk

For GATE version 1.5.1
Document Version: *Revision* : 2.1 Date: October 2000

Contents

Chapter 1

Introduction

1.1 LaSIE: Past, Present and Future

The LaSIE (Large Scale Information Extraction) system has been under on-going development in the Natural Language Processing Group, Department of Computer Science, University of Sheffield, since 1995. It was developed to illustrate a particular approach to Information Extraction (IE), though it was initially spurred into existence and shaped by the Sixth Message Understanding Conference (MUC-6), held in 1995 [4, 10]. Subsequent to this, the modules comprising the original LaSIE system (LaSIE 1.0) were "sawn" apart and re-integrated within GATE, to form a functionally equivalent, but architecturally very different, version of LaSIE (LaSIE 1.5). GATE, the General Architecture for Text Engineering, had been undergoing simultaneous development at Sheffield (see `http://gate.ac.uk`). GATE supplies document collection management facilities, a visual execution model, visualisation tools for viewing module results, and a uniform API to an underlying database, which serves both as a persistent store for module results and a communication medium via which module results are passed from one module to another.

To serve as an illustration within GATE of how language processing modules could be combined into application systems, a version of the LaSIE 1.5 system, called (inappropriately, with hindsight) VIE – Vanilla Information Extraction System – was bundled with GATE, and has continued to be so until the present. VIE is, to all intents and purposes, LaSIE 1.5.

Following the initial release of GATE and VIE, LaSIE continued to evolve and a refined version of the system was used for the Seventh Message Understanding Conference (MUC-7), held in 1998 [5, 15]. Referred to locally as LasIE-II (officially, LaSIE 2.0), this enhanced version of the system differed from the earlier one only slightly at the architectural level – in the position of the List Lookup module, and the separation of the Template Writer module from the Discourse Interpreter. However, many of the individual modules were refined in numerous ways, the most notable changes being the almost completely rewritten grammar (the original attempt to acquire a grammar from the Penn Treebank was abandoned) and the introduction of event coreference and a focus mechanism into the discourse interpreter.

Since MUC-7 (April, 1998) LaSIE has principally been used as a starting point for IE systems in new application areas (e.g. in bioinformatics where LaSIE has served as the basis for the EMPathIE and PASTA systems [13]) and for other text processing research into areas such as question answering (LaSIE has served as the basis of the Sheffield entries into the TREC-8 and TREC-9 QA track [16, 8]) and summarisation [1]. During this time the core

LaSIE system has remained largely unchanged. Some revisions have been propagated back into the base system from these other projects, but in essence the core IE system remains as it was entered into MUC-7. It is this system – LaSIE 2.1 – that is described in the current document, along with certain additions that serve to assist in development of new IE applications and in testing/debugging aspects of the system.

The future of LaSIE is unclear. The name ("large scale") is hardly appropriate today given current views on the volume of texts IE systems should be able to process as a matter of course (and LaSIE has never been the fastest of systems, though speed was never a focus of our research and little effort has been expended on optimising performance). Further, approaches to IE have changed and systems that presuppose a level of meaning representation independent of the text and rely upon manually assembled rule-bases are not in vogue. Nevertheless there are reasons to believe that LaSIE, or a descendent thereof, should have a future. Many of the components in LaSIE (section analyser, tokeniser, sentence splitter, gazetteer lookup) are highly reusable and required by virtually any approach to IE. The "meaning representation" approach to language processing, while under healthy critique from statistical approaches, is certainly not dead, and will continue to represent one strand of important research work in NLP. The manually built rule-bases which direct several of the system's key components consist of rules which may well be learnable in either supervised or unsupervised fashion, either fully automatically or semi-automatically, with a 'human in the loop'. Also, increased computing power means that while expectations have been raised about the overall volume of texts that should be processed, many important text collections can now be processed by systems like LaSIE in much more respectable times. Finally, there is something to be said about looking to the future with a stable, well-tested code base.

1.2 Documentation

This document describes LaSIE version 2.1 as embedded within GATE version 1.5.1. It is a technical specification document, not a user guide. It documents each of the modules in LaSIE in terms of: a functional overview, its interface to GATE (annotation input from or output to the GATE document manager), external processes the module may invoke, external resources on which the module depends, and known limitations.

This document does not describe GATE. A separate GATE User Guide [9] exists, as does a document describing how to integrate modules into GATE [3]. Full understanding of the present document will almost certainly require reference to these other documents as well.

1.3 LaSIE System Architecture

A schematic of the LaSIE system, showing the sequence of the main LaSIE modules is shown in figure 1.1.

1.4 CREOLE Modules in GATE

To take advantage of the facilities GATE has to offer (document management, visual execution, visualisation tools for module results, a uniform API to an underlying database for persistent storage of module results and communication of module results between modules)

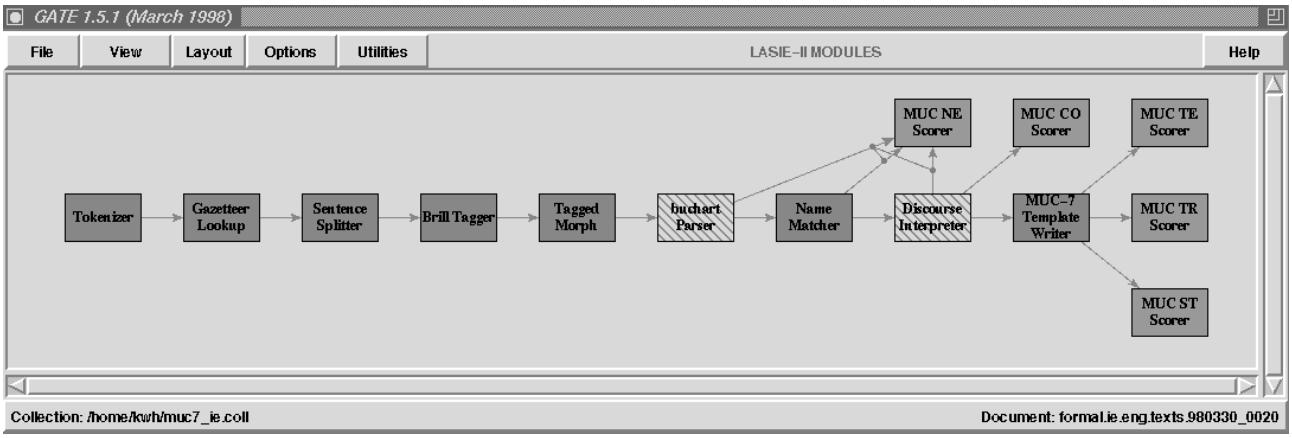


Figure 1.1: LaSIE Architecture

an application system such as LaSIE has to be broken up into a number of functional subcomponents each of which becomes a CREOLE (Collection of REusable Objects for Language Engineering) module within GATE. While this document does not describe how to do this in detail (for this see the CREOLE Developers Manual [3]), the reader of this document should be aware of some of the basic concepts behind the CREOLE approach. These are introduced here.

The motivation for CREOLE is to provide a set of encapsulated language engineering modules which can be reused in different applications, or for which functional equivalents can be easily substituted ('plug and play'). Three things are necessary for each CREOLE module:

configuration file The configuration file tells GATE about the module – what sort of input annotations and attributes it requires and output annotations and attributes it produces; what sort of viewers are required for its output; what parameters may be passed to the module. This information allows GATE to validate the modules position in the executable graph (see Figure 1.1) and to make appropriate viewer selection menus and parameter dialogue boxes available for the module. All configuration files are written in tcl and reside in a file called `creole_config.tcl` in the module's root directory.

wrapper The wrapper communicates between GATE and the module and serves to interface between modules, written in arbitrary languages, which were never designed to be included in GATE. The basic logic is the same for every wrapper. First, it extracts the input the module requires from the GATE document manager (GDM) – perhaps the text of the document to be processed and/or annotation information produced by earlier modules – and transforms this data into the native input form that the module requires (perhaps by writing to a temporary file). Next, the wrapper invokes the module passing it as input the information retrieved and transformed from the GDM. Finally, the wrapper takes the output the module produces, maps it into the appropriate annotation-based representation for the GDM and stores the output in the GDM. Since wrappers must communicate with the GDM via the GDM API, they must be written in a language that supports the GDM API – tcl and c++ for GATE 1.5 and Java for GATE 2.x. The wrapper (code and executable, if there is one) lives in the module root directory and must bear a name which is recorded in the configuration file.

module core The module core is the code which actually carries out the language processing task. As noted, it is invoked by the wrapper, which also arranges to pass it its input and to make use of its output. Module cores may be written in any language supported on the platform running GATE and may embody arbitrary functionality. Their top level executable resides in the module root directory, by convention, though they may organise file space below this point as they see fit.

CREOLE modules may be **tight coupled**, **loose coupled** or **dynamically coupled** within GATE 1.5. Tight coupled modules are modules whose wrappers and cores are written entirely in c++. They are compiled as part of the GATE executable. This offers speed, but means all of GATE must be recompiled whenever the module changes. Loose coupled modules are modules whose wrapper is interpreted at runtime (i.e. whose wrappers are in tcl), and whose core is run as a separate external process, executed by the wrapper (unless the module core code is also in tcl and resides in the same file as the wrapper). Dynamically coupled modules are modules whose wrappers are dynamically linked to the GATE executable at run time (and hence are in c++) and whose cores are either compiled into the same linked library as the wrapper or are executed as separate external processes by the wrapper.

1.5 Acknowledgements

LaSIE has been created by many people. These people had to eat while they were being creative, and for this they required money. Funding for people who have contributed to LaSIE over the years has most gratefully been received from:

- The UK Engineering and Physical Sciences Research Council (EPSRC). The core funding which supported the initial creation of LaSIE and GATE came from EPSRC grant GR/K25267. A new version of GATE is being developed with further support from the EPRSC (GR/M31699).
- The European Commission Telematics Programme (Framework IV) – the ECRAN, AVENTINUS and ELSE projects.
- GlaxoWellcome Research, in support of the EMPathIE project.
- Elsevier Science, in support of the EMPathIE project.

The initial LaSIE system was created by Rob Gaizauskas, Kevin Humphreys, Hamish Cunningham and Takahiro Wakao. Saliha Azzam, Mark Hepple, Chris Huyck, Brian Mitchell, Sandy Robertson and Pete Rodgers have all contributed significantly to subsequent developments. Yorick Wilks has provided ideas, benign criticism and ceaseless support. As we hope is clear from references in the following document we owe a significant debt, both intellectually and practically, to others in the NLP community who have supplied us with ideas and, in some cases with code.

Chapter 2

Sectionizer

2.1 Overview

The SECTIONIZER module scans texts for predefined start/end pairs of regular expressions, and creates a new annotation for each match. It is intended primarily to be used to detect gross section divisions in texts, and the annotations that result can be used by subsequent modules to determine what sorts of processing to apply to certain sections. In particular it can be used to identify sections to be excluded from processing (e.g. tables, figures, bibliographys). However, it is not limited to this purpose and can be used for finer grained analysis, e.g. to detect SGML tags (see the example in 2.2.5 below) or to recognise named entities.

It is not currently used in the LaSIE MUC system, which uses the tokeniser for much the same function, but can be added at any stage for the identification of text regions not captured by the TOKENIZER module.

2.2 GATE Interface

2.2.1 Input

Raw text, as a byte sequence.

2.2.2 Output

Annotations and attributes as specified in the regexp definitions file.

2.2.3 Resources

The module reads the definitions file at run time, by default called `sectionizer.def` and changeable through the module parameters. The definitions file contains entries of the following form, with each line containing all four fields:

```
annotation_type
{attribute1_type:attribute1_value ... attributeN_type:attributeN_value}
start_regexp
end_regexp
```

Within each regexp, the first brackets (opening for start_regxp, closing for end_regexp) delimit the annotation span.

2.2.4 Parameters

RegExp file The full path of a definitions file to be used at runtime. The file can be edited and the module reset and rerun from the interface to pick up changes, though only annotation types specified in the `post_conditions` entry of the `creole_config.tcl` file will be removed from the GDM database when the module is reset.

The `viewers` entry defined in the `creole_config.tcl` file, and used to control the contents of the module's results menu in the interface, is automatically updated each time the definitions file is read at runtime. An entry is created for each annotation type included in the file, with the GGI 'section' viewer used for annotations with no attributes, and the 'single_span' viewer for annotations with attributes.

2.2.5 Processing

Given the following entry in the definitions file:

```
sgml {type:end} "(</)" "(>)"
```

the module will search for the character sequence `</` then record the position of the opening bracket, at the start of the sequence, as the annotation start. The expression `"</(.)"` could be used instead to record the position of the character following the `</` sequence.

The second regexp is then used to search for the first `>` character following the position of the closing bracket in the first regexp match. The position of the closing bracket in the second regexp match is then recorded as the annotation end.

If both regular expressions are matched, a new annotation is created, in this case an `sgml` annotation with a `type` attribute, the value of which is `end`. The attribute list may be a space separated list of multiple attributes, or it may be an empty list.

The first regexp pattern is then searched for again, from the position of the closing bracket in the previous end regexp match, to attempt to create further annotations of the same type, before moving on to the next line in the definitions file and restarting searches from the beginning of the text. Multiple lines can be included in the definitions file for the same annotation types.

Each regexp in the definitions file is interpreted as a string and therefore requires tcl's single quoting of all special regexp characters within the string, and double quoting of these characters to force a non-special interpretation. See the tcl/tk manual pages for details of the special characters and their interpretation within strings.

One annotation type, called `exclusion_zone`, is treated specially by the module. If annotations of this type have already been created, possibly by the module itself from previous lines in the definitions file, subsequent regexp patterns will not be allowed to match text within the annotations. `exclusion_zone` patterns should therefore normally be placed first in the definitions file.

Chapter 3

Tokenizer

3.1 Overview

The TOKENIZER module identifies word boundaries in a text, returning byte offsets (or character positions) to be used as indices in the GDM database.

The TOKENIZER also attempts to identify certain known text types — currently New York Times, Wall Street Journal and Reuters — the formats of which allow the identification of a document identifier, header/body/trailer boundaries, and section (or paragraph) boundaries. For New York Times and Wall Street Journal texts, exclusion zones (as specified for MUC-6 and MUC-7) are also recognised, and a flag is set if `<s>...</s>` SGML sentence markup is found.

Unrecognised text types are treated as plain text, with no header/body boundary, and with blank lines assumed to indicate paragraph boundaries.

3.2 GATE Interface

The TOKENIZER is implemented via a set of regular expression patterns which are translated to `C` using `flex`. The file `.../creole/tokenizer/tokenize.ll` contains the patterns.

3.2.1 Input

Raw text, as a byte sequence.

3.2.2 Output

`token` annotations.
`section` annotations.
`header` annotation (one only).
`body` annotation (one only).
`trailer` annotation (one only).
`exclusion_zone` annotation (one only, with multiple spans).

`text_type` document attribute ("NYT", "WSJ", "REUTERS", or "plain").
`wsj_id` or `nyt_id` document attribute (identified using the `<DOCNO>` SGML markup).
`sentence_markup` document attribute (if `<s>` SGML markup is present).

3.2.3 Processing

The `flex` patterns define the following items to be tokens:

1. a contiguous character string in which each character is alpha-numeric or a hyphen ('-'), e.g. "Chicago", "old-fashioned", "1989".
   Hyphens are not included in character strings which occur within a recognised header section, and are instead treated as separate punctuation symbols.

2. punctuation symbols: ",", "$", ".", etc.

3. special cases involving apostrophes, e.g. "'s", "'d", "'ll".

4. compound negations: "mustn't", "wasn't", "couldn't", "aren't", etc.
   These are treated as single tokens (according to MUC-6 definitions) instead of being split into two (e.g. must+n't) or three (e.g. mustn+'+t) separate tokens. An explicit list of verbs which can combine with "n't" is used, mostly modal and auxiliary verbs. Irregular forms, such as "can't", "won't", "ain't", are listed in full.

5. words beginning with "O' ", as in "O'Keefe".

6. Wall Street Journal and New York Times SGML markups: `<DOC>`, `</DOC>`, etc.
   An explicit list of known SGML markups is used, rather than the simple recognition of `<>` and `</>` notation alone.

7. SGML symbols, such as `&LD;`, `&ndash;`, etc. Unlike the SGML markup, these are not explicitly listed and are identified solely from the format.

8. Wall Street Journal and New York Times document numbers: a numeric string starting with '8' or '9', followed by a hyphen and a four digit number, e.g. "940427-0051" (the `<DOCNO>` SGML markup is not used to identify this string).

Chapter 4

Gazetteer Lookup

4.1 Overview

The GAZETTEER LOOKUP module attempts to identify phrases and keywords related to named entities, as defined for the MUC tasks. This is done by searching a series of pre-stored lists of organisations, locations, date forms, currency names, etc., some of which have been derived from the gazetteer lists provided as part of the MUC training data. While originally created to support named entity identification strictly for the MUC Named Entity tasks (MUC-6, MUC-7), this facility has been extended to allow arbitrary lists of terms to be used and arbitrary tags to be assigned. As such it has proved useful for terminology tagging tasks in technical domains, e.g., for identifying protein names in molecular biology texts.

4.2 GATE Interface

The various lists are translated into a series of `C` finite-state recognisers using `flex`. These are compiled into GATE so that no external search through the original list sources is required.

4.2.1 Input

`token` annotations, optionally with `pos` attributes.

4.2.2 Output

`lookup` annotations, with `tag` and `type` attributes.

The `tag` attribute is used to indicate a major class for a match, such as organisation or location, and the `type` attribute specifies further distinctions within the class, if appropriate. For example, the location `tag` has several `type` subclasses such as city, province, country, region. The attribute values are based simply on which list a match was found in.

4.2.3 Resources

A set of lists of terms and keywords, one entry per line, together with a `lists.def` configuration file defining the `tag` and `type` values associated with each list, are processed at compile time into a single multi-stage recogniser. The set of lists used for the MUC-7 system is as follows. Most of the lists were compiled manually from various resources available on the web,

but where specific resources were used, these are indicated below. Aircraft and airport lists were introduced for the MUC-7 training task (air crash scenario), and spacecraft and satellite lists were introduced for the MUC-7 test task (rocket launches).

aircraft.lst aircraft manufacturers.

aircraft_names.lst aircraft model names and numeric codes.

airline.lst keywords which typically occur as part of airline company names.

airlines.lst airline company names.

airports.lst airport names, treated as locations for MUC.

cdg.lst company designators, such as "Ltd", "Corp", "Co", etc., based on the list provided as part of the MUC training data.

city.lst city names, taken from level 1 entries (basically country and state capitals) in the gazetteer provided as part of the MUC training data.

company.lst company names, originally taken from a list provided by the Consortium for Lexical Research.

country.lst country names, taken from level 1 entries in the gazetteer provided as part of the MUC training data.

country_adj.lst adjectives describing nationality, such as "English", "French", "American".

currency_unit.lst international currency names, taken from the MUC reference resources.

date.lst specific date terms such as "today" and "tomorrow", also including season names.

date_post.lst keywords used to modify dates and time periods.

date_pre.lst prefixes used to modify dates, such as "mid", "end of", etc.

datespan.lst keywords used to indicate time periods.

day.lst day names.

department.lst government departments.

festival.lst festival day names.

govern_key.lst keywords which typically occur as part of government organisations.

government.lst government organisations.

loc_key.lst keywords which typically occur as part of location names.

loc_prekey.lst prefixes used to modify location names.

ministry.lst government departments.

months.lst month names.

non_company.lst organisations which should not be marked as either company or government, e.g. "UN", "NATO", "EU".

nonspec_date.lst terms indicating time periods, which can used alone, e.g. "week", "decade"..

org_base.lst terms indicating organisation names, which can be used alone, e.g. "association", "union", "school".

org_key.lst keywords which typically occur as part of company names.

others.lst keywords and complete terms which are not of any category to be marked up, e.g. "Index", "World War I".

othorg_key.lst keywords which typically occur as part of non-company and non-government organizations.

person_ambig.lst person first names, ambiguous between male and female.

person_female.lst female person first names.

person_full.lst significant complete person names, e.g. "Bill Clinton".

person_male.lst male person first names.

planet.lst planet and moon names.

province.lst province and US state names taken from level 1 entries in the gazetteer provided as part of the MUC training data.

region.lst geographical regions.

satellite_comms.lst

satellite_destruct.lst

satellite_intell.lst

satellite_other.lst

satellite_research.lst

satellite_tv.lst

satellite_weather.lst satellite names, based on lists from NASA and other websites.

spacecraft_rocket.lst

spacecraft_shuttle.lst

spaceprobe.lst

spacestation.lst spacecraft names, based on lists from NASA websites.

times.lst terms indicating times of day.

timespan.lst units used for time periods.

timex_pre.lst prefixes used to modify time periods.

timezone.lst time zone names and abbreviations,

title.lst civilian person titles, such as "Mr" and "Dr", and including management position titles, such as "President" and "CEO".

title_mil.lst military personnel titles.

water.lst names of bodies of water, to be marked as locations.

The format of each line in the configuration file, by default `lists.def` as specified in the module's Makefile, is as follows:

```
list_filename:tag_value:type_value
```

For example,

```
city.lst:location:city
```

The `type_value` is optional, and if omitted the filename, less any `.lst` suffix, will be used for the `type` attribute. The configuration file is only used at compile time.

Additional regexp recognisers — `year`, `time_of_day` and `timex_trailer` — are also included in the module, mainly for the extended set of MUC-7 time expressions, and are not controlled by the `lists.def` configuration file describing the set of lists to use. These are effectively built-in lists, with fixed `tag` and `type` values, and the `lists.sh` lexer generation script, described below, must be modified to change the values or to remove the lexers.

4.2.4 Processing

The various lists, specified in a `lists.def` file, are translated into sets of regular expressions by the `lists.sh` script, which are then translated into a series of `C` finite-state recognisers using `flex`. These are compiled into a single `list_lookup` executable, so that no external search through the original list sources is required. The module effectively forks a separate GATE process, generating a temporary single document collection on which the `list_lookup` executable is run.

The module passes tokens to the recognisers, each with an associated offset range. The recognisers can match any single token or sequence of tokens, and the module creates new annotations using the offset ranges.

If no part-of-speech tag attributes are present on the token annotations, all token strings are passed to the finite state recognizers. Otherwise only token strings tagged as nouns and adjectives are passed in. The module can therefore run before or after the part-of-speech tagger module.

The longest match in each individual list, which may cover multiple tokens, is returned, but matches in different lists, of either the same string or a substring, will cause the addition of multiple annotations for the same text span.

For the MUC systems, matching is case sensitive, with one extension: not only will any all uppercase tokens in a text match their exact equivalents in a list; such forms are also converted to uppercase-initial before matching and hence will also match list entries in which only the first character is uppercase. For example, *FORD* in a text will match either `FORD` or `Ford` in a list, but not `ford`. This extension was added to deal with matching in headlines of newswire texts, which are frequently all uppercase. For the EMPathie and PASTA systems matching was altered further, so that if a list entry is lower case matching against it is case insensitive, but if it is upper or mixed case, matching is case sensitive (this was done because in these systems list matching was used for terminology identification, and terms, like common nouns and unlike proper nouns, can occur either in all lower or uppercase-initial forms depending on position in a sentence).

4.3 Maintenance: Addition of New Lists

To add lists for an additional class of terms, firstly decide how much structure the terms you want to identify have. There may be a clear division between 'head' words and 'modifiers' of those heads, e.g.: *alcohol dehydrogenase*, *alcohol oxidase*, *glycerol dehydrogenase*, and this is best dealt with by using separate lists, e.g. for proteins and protein modifiers, and then subsequent grammar rules in the parser (Chapter 8) to combine them.

Create the new lists in the `.../creole/lists` directory, by convention with a `.lst` suffix for each file:

```
protein.lst:
dehydrogenase
oxidase

protein_mod.lst:
alcohol
glycerol
```

Then add a new entry for each list in the appropriate `.def` configuration file (e.g. `empathie.def` specifies the lists to use for the EMPathIE application). Each line in the configuration file should be of the form `list_name:class:subclass` e.g.:

```
protein.lst:protein:head
protein_mod.lst:protein:modifier
```

If a match is found in a list, `class` will be used as the value of the `tag` attribute on the `list_lookup` annotation in the output, and `subclass` as the value of the `type` attribute. If `subclass` or both `class` and `subclass` are unspecified in the configuration file, then `list_name` (without the `.lst` suffix) will be used as the value.

To compile the list lookup module, including the new lists, do:

```
% make clean; rm *_a? *_a?.? *.a
```

to clear out all old lists, then

```
% lists.sh empathie.def
```

to generate flex input from the lists specified in the `.def` file, and finally

```
% make
```

which will take a while. (If a 'mk_lexer: not found' error is reported, ensure that your PATH includes '.').

If you only add new entries to existing lists, you can skip the first step to clear the old lists, and the compilation will be quicker.

You should then be able to rerun the lists module from GATE and use the results viewer (which displays both class and subclass) to check that list entries have been picked up correctly.

An important point regarding multi-word entries in the lists is that each token, as defined by the Tokenizer module, must be space separated. So, for example, *myo-inositol* (as a protein modifier) should be given as:

```
myo - inositol
```

in the lists file, if the tokeniser module classifies this as three separate tokens.

Chapter 5

Sentence Splitter

5.1 Overview

The SENTENCE SPLITTER module is based on the sentence splitting algorithm used in the Sussex MUC-5 system, POETIC [7]. It identifies sentence start and end byte offsets, making use of SGML sentence markup if present.

5.2 External Processes

5.2.1 Software Requirements

The SENTENCE SPLITTER is implemented as a `perl` script
(file: `.../creole/splitter/sent_split.perl`).

5.2.2 Input

Start and end byte offsets, one pair per line, each followed by the corresponding character string. Newlines from the raw text are also treated as tokens, written as `\n` with byte offsets.

5.2.3 Output

Byte offset pairs, one per line, representing sentence start and end positions.

5.2.4 Resources

The sentence splitter uses a list of common English words (file: `.../creole/splitter/wordlist`). It also make use of the external `morph` program (described in Chapter 7) as a stemmer so that only root forms are searched for in the wordlist file.

The `perl` script also contains a table listing common abbreviations for company designators and (English) person titles.

5.2.5 Parameters

An alternative wordlist file can be specified via the GGI. A flag can also be set to force sentence SGML markup to be used if present.

5.2.6 Processing

A sentence end is assumed if:

1. a period is preceded by a non-abbreviation and followed by an uppercase token whose root (obtained via the `morph` program) is a common word (i.e. is in the wordlist file);
2. a period is preceded by a single character token which is preceded by a lower case token, unless the period is followed by a token whose root is not a common word (to allow for cases such as "B. Clinton" or "U.S.");
3. a period is followed by " or ' or a newline;
4. a newline is followed by a `</p>` (paragraph end) SGML tag;
5. an SGML tag is both preceded and followed by a newline;
6. a newline is followed by a "@" table marker in a Wall Street Journal text.

If the flag to use SGML markup is set, and the markup is present, the input is simply searched for `<s>...</s>` pairs, and the corresponding offsets are written out.

5.3 GATE Interface

5.3.1 Input

`token` annotations.
`lookup` annotations (optional).
`sentence_markup` document attribute (optional).

5.3.2 Output

`sentence` annotations with a `constituents` attribute, the value of which is an ordered list of GDM database identifiers of the `token` annotations within the sentence span.

5.3.3 Processing

`token` annotations are written out to a temporary file as a byte offset pair followed by the string within the span. If a newline occurs between the spans of two adjacent `token` annotations, this is written out as if it were a token.

At each token position, if any list lookup annotation is found at that position, the string corresponding to the list entry is written out as a single unit for the external sentence splitter `perl` script, within which no sentence boundary can be proposed. If multiple list lookup annotations are found at the same position, the entry with the longest span is used. This avoids the proposal of sentence boundaries within list matches.

The external `sent_split` script is then called, with a `-s` (SGML) flag if the `sentence_markup` document attribute is present, and the script's output is used to create new `sentence` annotations, recording, as an attribute, the token i.d. numbers within the sentence span. This representation, as opposed to an annotation without attributes, is adopted to avoid repeatedly calculating the same sets of `token` annotations within each sentence in subsequent modules.

5.4 Limitations

The SENTENCE SPLITTER is reasonably domain independent, although the current lists of common abbreviations may need to be extended for non-financial domains.

Chapter 6

Brill Tagger

6.1 Overview

The Brill tagger [2] is a rule-based part-of-speech tagger that has been extensively trained on the Penn TreeBank corpus of manually tagged Wall Street Journal texts [18]. It uses, therefore, the 48 part-of-speech tags which make up the Penn TreeBank tag set.

The original tagger resources have been modified slightly for use in MUC. The modifications include the introduction of new tags for dates, SGML markup, and punctuation symbols, and the addition of several new lexical and contextual rules to the original rule base.

6.2 External Processes

```
/* Copyright @ 1993 MIT and the University of Pennsylvania */
Written by Eric Brill, brill@blaze.cs.jhu.edu
```

6.2.1 Software Requirements

The Brill tagger is public domain software, available from:
`ftp://ftp.cs.jhu.edu/pub/brill/Programs`
It is written in `C` with a number of training scripts in `perl`.

6.2.2 Input

The Brill tagger expects a plain text file as input, formatted with one sentence per line.

6.2.3 Output

The output is a version of the input file with a part-of-speech tag appended to each token, e.g.:

```
One/CD of/IN the/DT differences/NNS between/IN Robert/NNP L/NNP ./PERIOD James/NNP
```

6.2.4 Resources

The Brill tagger requires four resource files:

Lexicon

File: `.../creole/brill/MUC6/LEXICON.MUC6`

The original lexicon file (produced from training on the Penn TreeBank corpus) was modified for MUC-6 to include additional tags:

`DATE` (days of the week, month and season names)
`SGML` (markup tags occurring in Wall Street Journal articles)
`PPS` (possessive personal pronouns)

Irregular negated modals ("can't", "won't" and "ain't", as identified by the TOKENIZER module) were also added to the lexicon file as normal modals (`MD` tag).

Several punctuation symbols whose tags were the same as their surface forms were assigned the existing `SYM` tag. Separate tags were used for the `COMMA` and `PERIOD` punctuation symbols.

Lexical Rules

File: `.../creole/brill/MUC6/LEXICALRULEFILE.MUC6`

The original lexical rule file was supplemented with a rule to tag "5" as a cardinal number (CD tag), the omission of which was presumably just an effect of the original training data used. With the default lexical rules "5" is tagged as a normal noun (NN tag).

Contextual Rules

File: `.../creole/brill/MUC6/CONTEXTUALRULEFILE.MUC6`

For the contextual rule file, the following changes were made:

- The default part-of-speech tag for "marks" is changed from a verb (`VBZ` tag) to a plural noun (`NNS` tag) when preceded by a number (`CD` tag), to allow the recognition of German currencies (for the MUC-6 Named Entity task).
- Several rules to tag "fall" as a noun (`NN` tag) were added, to allow its recognition as a date (for the MUC-6 Named Entity task).
- The default part-of-speech for "operating" is changed from a verbal gerund (`VBG` tag) to an adjective (`JJ` tag) when it precedes a noun. This permits the recognition of, e.g., "operating officer", at the GAZETTEER LOOKUP stage (required for the MUC-6 Scenario Template task) as only nouns and adjectives are passed on to this stage.
- Several rules were added to change various tags to proper nouns (`NNP` tag) when followed by either a company designator or a period.
- A preposition (`IN` tag) followed by a coordinator (e.g. "and") (`CC` tag) is changed to an adverb.

Bigrams

File: `.../creole/brill/MUC6/BIGRAMS`

The default null bigram file was left unchanged.

6.2.5 Parameters

Alternative resource files can be specified via the GGI. An additional lexicon file can also be used — see the tagger's own documentation for details of the file formats.

Also settable from the GGI are parameters to run only the start-state-tagger process, to dump output after this stage during full processing, and to tag texts a certain number of lines at a time if memory is insufficient to process a full text — again, see the tagger's own documentation for further details.

Enable Tag Correction This parameter activates several heuristics intended to correct common mistakes made by the Brill tagger.

Firstly, if any lower case tokens are tagged by Brill as either NNP or NNPS (proper noun), the tag for the token is converted to JJ (adjective).

Secondly, each upper case token in the document header which has been tagged as a proper noun, is tested to attempt to determine whether it should be retagged as a common noun. If a List Lookup entry is found beginning at the same position as the token, the tag is left as proper noun, reflecting the fact that the Lists contain mainly proper nouns, though this is not true of all list categories. If no List Lookup entry is found, the token is checked to see whether it is a known common word. This is done by calling the Morph module's external executable, called `morph`, with the token string to find the root form, and then by matching against the list of common words used in the Sentence Splitter module. If a match is found, the token's tag is converted to NN (common noun).

Filter Gazetteer Entries This parameter acts to remove single token List Lookup entries, if Brill has tagged the corresponding token as anything other than either NN, JJ, RB or DATE. This will remove, for instance, the person name match for the token 'Sue', if Brill has tagged the occurrence as a verb.

As a result of this potential modification of the List Lookup module's results, resetting the Brill tagger from the GATE interface will also reset the List Lookup module, to force the regeneration of the original set of list matches.

6.2.6 Processing

The Brill tagger's own documentation is distributed with GATE in:
`.../creole/brill/RULE_BASED_TAGGER_V1.14/Docs`
Publications related to the Brill tagger can be found at:
`ftp://ftp.cs.jhu.edu/pub/brill/Papers`

6.3 GATE Interface

6.3.1 Input

`token` annotations.
`lookup` annotations (optional).
`sentence` annotations.

6.3.2 Output

`pos` attributes on the existing `token` annotations. The attribute values are upper case tag names, as output by the tagger.

6.3.3 Processing

The text strings within the span of each `token` annotation are written out to a temporary file, separated by single whitespaces. A newline is written out after the current token if it is at or beyond the current sentence end. The Brill tagger is then called with this temporary file as input.

The list of tags output from the Brill tagger is read in and each one added as the value of the `pos` attribute for the next `token` annotation. The tagger may output alternative tags for a single token in the input, but in this case only the first is added as the value of the `pos` attribute.

6.4 Limitations

The Brill tagger is domain dependent because its default resource files were produced by training on Wall Street Journal texts. The tagger can also be used in other domains with the same lexicon and rule sets, but presumably gives a reduced performance. It is possible to retrain the tagger given a new corpus of hand-tagged texts.

The ability to retrain also means that the Brill tagger can be used in languages other than English, given a suitable corpus of hand-tagged texts.

Chapter 7

Morph

Both the MORPH and TAGGED MORPH modules in the GGI use the same external process and GATE interface, with a flag to indicate which version is required.

7.1 Overview

The MORPH module is in fact a lemmatiser rather than a full morphological analyser. However, in addition to producing a root form for each token it is given, it also produces a normalised affix (e.g. "ed" for all past participle forms, both regular and known irregulars).

The TAGGED MORPH version only processes noun and verb tokens, as indicated by part-of-speech tags, using the tags to increase accuracy, and avoiding irrelevant tokens to increase processing speed.

7.2 External Processes

7.2.1 Software Requirements

The external morph program is implemented as a set of regular expression patterns which are translated to a `C` finite state recogniser using `flex`.

7.2.2 Input

Plain text on standard input with each token optionally prefixed by `begin(VERB)` or `begin(NOUN)` to restrict the set of applicable rules.

7.2.3 Output

A modified version of the input with each token changed to a `root+affix` format, if possible.

7.2.4 Resources

The set of regular expressions used represents both rules for the analysis of regular morphology and a list of exceptions. The exception list was originally derived from the exceptions used in WordNet [19], but this has been significantly revised following the analysis of a number of English corpora carried out by John Carroll (`johnca@cogs.susx.ac.uk`).

7.2.5 Processing

The regular expression rules implement their own tokenisation scheme, to allow the processing of plain text input. In the absence of prefixes to restrict the set of applicable rules, verb rules will be preferred in the case of multiple possible analyses. For example, the token "lives" will by default be analysed as "live+s" rather than "life+s", unless it is prefixed by `begin(NOUN)`.

7.3 GATE Interface

7.3.1 Input

`token` annotations, for the MORPH version.
`token` annotations with `pos` attributes, for the TAGGED MORPH version.

7.3.2 Output

`root` and `affix` attributes on the existing `token` annotations. Values for the `affix` attribute will be either `s`, `ing`, `ed`, `en`, or an empty string.

7.3.3 Processing

For the MORPH version, each token is written out to a temporary file, one per line to avoid the application of the external process's own tokenisation rules. For the TAGGED MORPH version, each token with a noun or verb `pos` attribute is written out, again one per line, but with the appropriate prefix.

The external `morph` process is then called with the temporary file as input, and the output parsed to split `root+affix` strings and assign each part as attribute values for the corresponding `token` annotations.

7.4 Limitations

The module is domain independent, the exception list having been developed from a number of corpora. Both the rules and exception list are obviously language dependent. Certain distinct English verbs are homomorphic in certain but not all inflectional forms and will confuse MORPH. For example, *ground* is both the base form of the verb *to ground* and the past and past participial form of *to grind*. Even TAGGED MORPH when given *ground* and told it is a verb (as opposed to the noun *ground*) will not be able to distinguish these cases, and will always get one of them wrong (the default is to treat the verbal form *ground* as `grind+ed`).

Chapter 8

buChart Parser

8.1 Overview

The parser is a modification of the Gazdar and Mellish bottom-up chart parser [12]. It uses a feature-based unification grammar to perform bottom-up phrase structure syntactic analysis of each sentence in its input and during parsing a semantic representation of each constituent is constructed compositionally from the semantic representations of the constituent's components.

The parser is complete in the sense that every analysis licensed by the grammar is produced, though there is a mechanism to control this. On completion a 'best parse' algorithm is run to select a single analysis of the sentence, which may be partial if no tree spanning the whole sentence can be constructed. The parser produces three sorts of output for each sentence in its input: a phrase structure analysis of the sentence, a semantic interpretation of the sentence, and an identification of certain specified categories of multiword names ("named entities") or terms in the sentence.

The parser operates in two main stages, first applying a sequence of specialised subgrammars to construct named entity or compound term noun phrases of specific semantic classes, then applying a sequence of general phrasal subgrammars. Each subgrammar can be run individually in the 'cascaded' mode of the parser.

8.2 External Processes

8.2.1 Software Requirements

The parser is written in Prolog and requires SICStus Prolog 3#6 or SWI Prolog 2.9. If SICStus is not found, the overall GATE build may fail to automatically create the `buchart` executable Prolog saved state and the `mkparser_sicstus` or `mkparser_swi` script must be run manually, after specifying the path to the appropriate Prolog executable.

8.2.2 Input

A Prolog readable file containing a single `doc_descriptor/1` term, specifying a document identifier for use in output files, followed by a series of `chart/3` terms, one per sentence. Each chart term is of the form:

```
chart(sentence_n:S,
      edges:Edge_List,
      next_edge_number:NE).
```

where S and NE are positive integers and Edge_List is a list of edge/10 terms, each of the form:

```
edge(Start_Token,
     End_Token,
     Category,
     Category_List,
     Creator_ID,
     Child_ID_List,
     Level,
     Start_Offset,
     End_Offset,
     ID)
```

In the initial input, Category_List, Creator_ID and Child_ID_List entries will be empty, and all edges will have Level 1.

Chart Edge Features

Each Category entry has the form:

```
Category(Feature1:Value1,...,FeatureN:ValueN)
```

where the number and type of features is dependent on the category type. The first four features are the same for all categories: s_form (surface form), m_root (morphological root), m_affix (morphological affix) and text (with a value of either header, body (default) or trailer if annotations of these types are present for the current text).

Nominal and verbal categories will also have person and number features; verbal categories will also have tense and vform features; and adjectival categories will have a degree feature. The list_np category has the same features as other nominal categories plus ne_tag, ne_type and gender.

The features expected by the buchart executable are specified by the feature_table/2 predicate at the top of the file compile_grammar.pl. The features produced by the wrapper for input to the executable are specified in a POS-tag to syntactic-category and feature-value table in either buchart.tcl or buchart.cc, depending which version of the wrapper is being used. From this, the POS tag NNS, for example, will be mapped onto the category n with the features number:plural, person:3). Clearly, both the Prolog feature_table and the wrapper table should be kept in correspondence.

8.2.3 Output

The executable writes out the details of one annotation per line to the standard output, which are read by the wrapper and used to create new annotations. Each line consists of an annotation type (name, syntax, or semantics), followed by a byte offset pair, and at least one value.

For `semantics` output the value is a Prolog readable list, containing the semantic representation of the sentence – unary and binary predicates produced from the compositional rules associated with the grammar rules. All NPs and VPs lead to the introduction of a unique instance constant, or ID number, in the semantics which serves as an identifier for the object or event referred to in the text – e.g. *company* will map to something like `company(e22)` in the semantics and *hired* to `hire(e34), tense(e34,past)`. Each of these instance constants is assigned a `realisation` property in the semantic representation, indicating, as byte offsets, the position in the text from which the semantics were derived. Nouns used as possessives or qualifiers also produce `realisation` properties. These `realisation` properties are required by the discourse interpreter (chapter 10) to enable it to annotate coreference chains in the text.

For `name` outputs, which are the results of the Named Entity parse, there are two values: a type, and the corresponding ID number in the `semantics` value. The type is taken from the first unary predicate in the semantics list which includes the name's ID number. (This can be modified by restricting the `ne_tag/3` definition in `buchart_io.pl`, e.g. to output only names of types required for the MUC NE task).

For `syntax` output the values are a syntactic class plus "constituents X", where X is an integer specifying the number of children of the current class to follow in the output stream. This information is used in the wrapper to construct `constituents` attributes linking different `syntax` annotations into a tree structure. If the number of children is 0 then the annotation is treated as a leaf node with no `constituents` attribute. An additional `root` attribute is also added for each new `syntax` annotation which does not represent the child of any other annotation.

8.2.4 Resources

A top-level `load.pl` file, in the `.../creole/buchart/grammar` directory, is used at compile time to specify which grammars, and in which order, should be compiled and built in to the `buchart` executable Prolog saved state. Each grammar file must specify which syntactic categories can be included in the best parse, e.g.:

```
% Best Parse Categories
best_parse_cats([ne_np]).
```

A special value of `best_parse_cats(all).` causes all inactive edges from the final chart to be passed on, and the best parse selection is not used in this case. The first NE subgrammar, `general_ne_rules.pl`, currently uses this special value to pass on all candidate sequences of proper nouns.

A `filter_chart.` flag can also be set in a grammar file. This has the effect of running the parser to completion with the subgrammars up to and including the grammar in the file in which the flag is set, then running the best parse selection mechanism (see 8.3.3 below) and passing on only a single analysis to the next subgrammar, removing all edges below the best parse edges. Without the flag, no edges are removed and the next subgrammar runs on the whole chart. The flag is currently set on the last of the NE grammars, to prevent the sentence grammar rules from reanalysing any NPs proposed by the NE rules. It is also set on the the last phrasal grammar (`s_rules`), but the best parse selection will be run anyway after the final subgrammar. Use of `filter_chart` can, and most likely will, make the parser incomplete with respect to the grammar. However, it allows the grammar writer to encapsulate grammars in

a fashion that minimises unexpected interaction effects (by blocking reanalysis inside phrase types found by earlier subgrammars) and leads to significant performance gains.

The format of each grammar rule is basically as follows:
`rule(a(features), [b(features), c(features)]).`
which should be read as a context free grammar rule of the form:
`a(features) --> b(features) c(features).`

In addition to the features included in the input edges (described above), `edge` and `sem` features are introduced within the grammar rules. `sem` uses a lambda style notation (with `^` as lambda as in [20]) to build up a QLF (quasi-logical form) style semantic representation, and `edge` is assigned a value of the form `offsets(StartN,EndN)` whenever an edge is completed (to specify the text span covered by the edge).

Named Entity Grammar Rules

The NE grammar for MUC-7 is split into 9 subgrammars: `general_ne_rules`, `aircraft_ne_rules` `person_ne_rules`, `location_ne_rules`, `space_ne_rules` `organ_ne_rules`, `money_ne_rules`, `time_ne_rules` and `timex_ne_rules`. The aircraft and space grammars were added for the MUC-7 ST training and evaluation tasks, and the timex grammar for the extended set of temporal terms introduced in the MUC-7 NE task.

An additional NE subgrammar, `default_ne_rules`, has also been added for MUC-7, which runs after the best parse selection flagged by the timex grammar. This applies various rules to any remaining unclassified proper noun sequences, attempting to use the context of any surrounding NEs which have been classified. This subgrammar also includes the `filter_chart` flag and so the best parse selection is run again to produce a final set of NEs.

Phrasal Grammar Rules

A sequence of 7 subgrammars make up the phrasal grammar: `npcore_rules`, `pp_rules`, `np_rules`, `vpcore_rules`, `vp_rules`, `rel_rules`, and `s_rules`. As in the NE grammar, the order of compilation is specified in the `load.pl` file, and the order of submodules specified in the `creole_config` files. The best parse categories for these subgrammars currently include all the categories from the preceding subgrammars, with the final `s_rules` subgrammar specifying the top level categories of all the preceding ones.

Some rules are duplicated in the subgrammars, such as PP attachment rules in the NP grammar as well as the PP grammar. In the cascaded mode of the parser, where each subgrammar is run separately, the duplicated PP rules allow attachment to any newly created NPs in the NP grammar, which would not be possible otherwise since the rules from the PP grammar would not be available at this point in the cascaded mode. Duplicate grammar rules will be removed during compilation of the subgrammars for the multiple grammar mode, and all rules will be available simultaneously in each complete grammar in this mode.

8.2.5 Parameters

The parser can be run either from the command line, as it is called by the GATE wrapper, or in interactive Prolog mode useful for tracing and debugging. Both modes take the following options:

-v write progress messages to standard output.

-d write debugging information to standard output.

-g `filename` compile and parse with a single grammar file, ignoring any other built-in grammars.

-f filter the final chart, even if the flag is not included in the grammar file specified by the `-g` option.

-ne run only the first of multiple compiled grammars, assumed to be the Named Entity grammar.

-p `filename` write a tree representation of the best parses to a file.

-b `filename` write a bracketed representation of the best parses to a file (suitable for scoring using the `evalb` utility).

-c `filename` write the final state of the chart to a file.

-o `filename` redirect standard output to a file.

A final argument given to the parser is assumed to be the input chart file to be parsed. Input files for particular texts can be obtained by running the parser from the GGI interface, then interrupting the Prolog process and copying the most recent file with an `_in` suffix from the `/tmp` (or `/var/tmp`) directory.

The interactive Prolog mode, for SICStus, is started as follows (`%` indicating the shell prompt, and `?-` indicating the Prolog prompt). The input file is the only compulsory argument.

```
% sicstus -l mkparser_sicstus.pl
?- parse(['-v','-o',output_file,input_file]).
```

8.3 Processing

On reading the input, each `chart` term is processed independently. The list of `edge` terms, representing the chart for a single sentence, is extracted, sorted, and parsed with each required grammar. After parsing with each grammar, the best parse selection mechanism is run to extract a new set of edges, which are then either sorted and passed on to the next grammar for further parsing, or used to write out final results.

8.3.1 Grammar Compilation

All grammars are used in a compiled form, where the compilation involves the expansion of each category's feature list to include all possible features for that category, using the feature table specified in `compile_grammar.pl`. The compilation also reverses the left-to-right ordering of categories, for more efficient access during parsing. Each compiled grammar rule includes an identifier for the compound grammar of which it is a part (e.g. `grammar3`), and each compiled rule is also assigned an ID number, assigned incrementally and unique within its compound grammar, which is included in any edges created by the rule during parsing and used for determining rule precedence within the best parse mechanism.

The compiled version of all the grammars is written out to the file `compiled_grammar.pl` at compile time. This file is currently only used in the creation of runtime binaries for SICStus PROLOG and can be safely deleted otherwise.

During compilation of the Prolog saved state, subgrammars listed in the `buchart/grammar/load.pl` file are compiled in sequence into a series of compound grammars. A new compound grammar is started after each `filter_chart` flag in a subgrammar, currently producing three separate grammars: NE, default NE, and Phrasal. The best parse categories of the final subgrammar in each compound grammar are used for the compound grammar itself.

In cascaded mode, all compiled-in rules are removed and a single grammar, specified with the `-g` command line option, is loaded and compiled at runtime.

8.3.2 Rule Application

Rules are selected from each successive grammar using the grammar identifier assigned during compilation. New edges created during parsing are assigned Level 2, and each new edge records the identifier of the rule used to create it.

Before each newly created edge is added to the chart, a check is made for any equivalent edges already existing in the chart. If one is found, the new edge is not added. Equivalent edges are required to have unifiable categories, with the exception of the `sem` and `s_form` features. The `s_form` values are not restricted at all, but different semantics produced by an earlier rule in the grammar are treated as equivalent. Rule ordering is determined using the identifier values assigned during compilation, with the effect being that equal edges with differing semantics will only be added to the chart in the same order as the rules to create them occur in the original grammar files. This edge ordering is then used by the best parse selection mechanism to prefer edges added by later grammar rules (i.e. later in the same grammar rule source file or later in the sequence of grammar rule source files).

When parsing with one compound grammar completes, the best parse algorithm is applied to all Level 2 edges, and its selection is then sorted and converted to Level 1. All edges are then cleared from the chart and parsing restarts with the new set of Level 1 edges and the next compound grammar.

8.3.3 Best Parse Selection

The best parse selection is made as follows: extract the set of shortest sequences of maximally spanning, non-overlapping edges of the best parse categories for the current grammar. When searching for the maximally spanning edge at each position, only the first of any set of equally spanning edges is used. This corresponds to the most recently created edge, again making the ordering of rules in the grammar files significant.

The algorithm steps through each position in the chart, searching for the longest edge from that position, and then continuing from the end of that edge. The best parse from each position is recorded and then retrieved if another path reaches the same point, to avoid duplicated searching. The best overall parse is then the one assigned to position 0.

If the `filter_chart` flag is set for the current grammar, the best parse, in addition to the current best parse categories, will return leaf edges, from the original input, for any spans not covered by a best parse category edge. The best parse in this case will have full coverage, rather than leaving gaps of uncovered input as in the normal case.

An additional test for the NE grammar (assumed to have the identifier `grammar1`) is also included to favour any edges with feature `source:list`. This is assigned by certain grammar rules to indicate that a complete NE was matched directly by a list entry in the gazetteer

lookup stage, rather than being constructed by a NE grammar rule and assigned some other, less certain, semantic class.

8.3.4 Semantics Translation

The `clean_semantics/2` function acts to translate the internal semantic representation built by the grammar rules into the output form shown here for the phrase "stepping down as chief executive officer":

```
step_down(e58),
tense(e58,present),
realisation(e58,offsets(225,230)),
as(e58,e60),
title(e60,'chief executive officer'),
realisation(e60,offsets(228,230)),
```

The internal representation for the above fragment, as passed to the `clean_semantics` function, is:

```
E1^[[compound(['step', '_', 'down']), E1],
    [tense, E1, present],
    [realisation, E1, offsets(225,230)],
    E2^[[as, E1, E2],
        [title, E2, ['chief',['executive officer']]],
        [realisation, E2, offsets(228,230)]]]
```

A new identifier is generated for each `^` variable (lambda abstracted), and then the list is descended, converting all 2 and 3 element lists into semantic types and attributes, respectively, with `compound` and list values for particular attributes treated specially as described below. Any variables which remain after all identifiers have been generated are instantiated with the string `'MISSING'` for debugging purposes.

Compound predicate names must be explicitly specified within the compositional semantic rules on the grammar rules, using the form:

```
compound(['String1','String2',...,'StringN'])
```

in place of the predicate. The strings will be concatenated into the full predicate name for use in the output semantics. The `compound` terms may be embedded.

A similar operation is also carried out for certain attributes which take strings as their values. In the semantic rules, these attributes may have, possibly embedded, lists of strings as values, to be concatenated into a single string for output. For example, an organization name grammar rule might specify that an organization name may consist of a proper name followed by a company designator, and the value of the `name` attribute in the `semantics` feature of the mother category would be simply a list consisting of the surface form features of the daughters:

```
%% ORGAN_NP --> NAMES_NP CDG_NP
rule(organ_np(sem:E^[[company,E],[name,E,[F1,' ',F2]]]),[
        names_np(s_form:F1),
        cdg_np(s_form:F2)
```

So, taking *IBM Corp.* as an example, the value of the `name` attribute in the `sem` feature of the above rule becomes `['IBM',' ','Corp']` and the final semantics translation process converts this to `'IBM Corp.'`.

The set of attributes for which such string concatenation will be performed must be explicitly declared in the `clean_semantics/2` function in the file `semantics.pl`. The current set is: `name, head, head1, head2, det, adj, pronoun, count, more, less, title`.

8.4 GATE Interface

8.4.1 Input

`token` annotations with `root` and `pos` attributes.
`sentence` annotations, with a `constituents` attribute.
`lookup` annotations, with `tag` and `type` attributes (optional).

8.4.2 Output

`name` annotations, with `type` and `id` attributes.
`syntax` annotations, with a `category`, `constituents` and `root` attributes.
`semantics` annotations, with a `qlf` attribute.

8.4.3 Parameters

A simple tree or bracketed list representation of the final 'best parse' can be dumped to a file, as can the final state of the chart.

'Verbose' and 'Debugging' flags can also be set to produce additional information in the 'best parse' output files.

In the submodules used in the cascaded mode, an additional 'Grammar File' parameter is also used, to specify the file to be compiled and run.

8.4.4 Cascaded Mode

The `.../creole/buchart` directory contains separate subdirectories for each grammar file, representing submodules which can be run in sequence in GATE to give a 'cascaded' mode. The submodules must be loaded by GATE at startup by setting GATE_CREOLE_PATH to include the `buchart` directory.

The order in which modules appear in GATE is controlled by the preconditions entry in the `creole_config.tcl` files. In this mode, the best parse selection is run after each subgrammar to select the output for the module, even if the `filter_chart` flag is not set, because the single subgrammar is the final one. However, edges below the best parse are still passed on to the next module, unless the flag is set.

The modified chart is passed on to each successive submodule through temporary files (in `/tmp`) which include the name of the submodule which created them. Each submodule uses the `pre_conditions` entry in its `creole_config.tcl` file to determine the name of the input file to use, assuming the last document attribute to be the name of the preceding submodule. The `init_buchart` module must be run first in the cascaded mode to write the initial chart from the GDM database.

The cascaded mode is considerably slower than the standard mode, particularly the write out of the initial chart done by the loose coupled `init_buchart` submodule. The mode is intended for the debugging of individual grammar files on specific short examples rather than full texts.

8.5 Maintenance: Adding a New Grammar or Grammar Rule

Generally each major terminology and phrasal category has its own grammar file, so, for example, to add a grammar for a terminology category such as "protein names", create a file named protein_ne_rules.pl in creole/ne_buchart/grammar with the following header:

```
:- multifile best_parse_cats/1, rule/2.
:- dynamic best_parse_cats/1, rule/2.
best_parse_cats([ne_np]).
```

The ne_np specified here is the name of the top-level category that this grammar produces and which should be passed on in the output. Now suppose we want to add a simple rule to say that a protein name = a protein modifier followed by a protein head, i.e.:

```
protein -> protein_mod protein_head
```

in standard grammar rule notation (the same as sentence → noun_phrase verb_phrase). Writing this in the PROLOG notation used in buchart is basically:

```
rule(protein, [protein_mod, protein_head]).
```

but we need to expand each of these categories. Each item found in the list lookup stage will be passed into the parser as:

```
list_np(ne_tag:class,ne_type:subclass)
```

i.e., its category will be 'list_np' with the features 'ne_tag' and 'ne_type', having the values 'class' and 'subclass' respectively, so the RHS (right hand side) of this grammar rule (the list in square brackets) should be:

```
[list_np(ne_tag:protein,ne_type:modifier),
 list_np(ne_tag:protein,ne_type:head)]
```

The head category, the LHS (left hand side), to be passed on then needs to specify the type of thing built by this rule. This is done using the `sem` (semantics) feature, which is passed on to be processed later. The value of the `sem` feature uses a special notation (called lambda abstraction) to introduce a new identifier with a list of information about that identifier. For terms, the only relevant information is the term class and the range to be highlighted (tagged) in the results, and so the top level (LHS) for most names will just be:

```
ne_np(edge:Edge,sem:E^[[type,E],[ne_tag,E,Edge]])
```

where `type` is the semantic category of the name class, e.g. `protein`, `edge` is a special feature which specifies the text range covered by this rule (if it matches), and the variable (capitalised) `Edge` just passes the value of this feature into the value of the `sem` feature.

So, the full rule for `protein` → `protein_mod protein_head` is:

```
rule(ne_np(edge:Edge,sem:E^[[protein,E],[ne_tag,E,Edge]]),
    [list_np(ne_tag:protein,ne_type:modifier),
     list_np(ne_tag:protein,ne_type:head)]).
```

Similarly, a rule which allows a protein head alone to be a complete protein (`protein` → `protein_head`) would be:

```
rule(ne_np(edge:Edge,sem:E^[[protein,E],[ne_tag,E,Edge]]),
    [list_np(ne_tag:protein,ne_type:head)]).
```

The first rule here would always be preferred (if it matches) by the best parse selection mechanism because it covers more text (i.e. has a longer length) than the second.

To integrate the new grammar into the parser, the file `buchart/grammar/load.pl` must be modified to include the new protein_ne_rules.pl grammar. The order here is important since if rules from two grammars classify the same range of text differently, the rule from the later grammar in the list will be preferred.

To recompile the parser, including the new grammar, run the `mkparser_swi` (for SWI PROLOG) or `mkparser_sicstus` (for SICStus PROLOG) script in `.../creole/ne_buchart`. Look out for reports of syntax errors from the grammar files. Now you can run the parser from GATE (you don't need to restart GATE, just reset buchart (right mouse button) and rerun) and check the output. It is better to develop grammars by running on short texts with simple lists of terms you want to recognise. The syntax results viewer of the parser module will then highlight everything that has been assigned a tree structure by a grammar rule, and clicking on the highlighted text will show the tree.

Chapter 9

Name Matcher

9.1 Overview

The NAMEMATCH module does not recognise new proper names but adds identity relations between those found by the parser. It may also assign a type to unclassified proper names, using the type of a matching name.

The matching rules are only invoked if the names being compared are either both of the same type, i.e. both already tagged as (say) organisations, or one of them is classified as 'unknown'. This prevents an already classified name from being re-categorised.

9.2 GATE Interface

9.2.1 Input

`name` annotations, with an `id` attribute.

9.2.2 Output

`matches` attributes added to the existing `name` annotations.

9.2.3 Resources

A table is used in `.../creole/namematch/name_lookup.h` to record non-matching strings which in fact 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.

9.2.4 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.

The following rules are applied to organisations (O), persons (P) or locations (L), or to more than one and possibly to all (A). They are not necessarily applied in this order.

1. If the two names are listed as equivalent in the lookup table of non-matching strings, then they are equivalent, e.g. "Coca-Cola" and "Coke" (A);

2. Does adding a possessive to the last token of one of the names cause a match? E.g. "Standard and Poor" is equivalent to "Standard and Poor's", or to "Standard's"';

3. Do all of the individual tokens (other than punctuation marks) match? E.g. "Smith, Jones" will match "Smith Jones". This is case-independent (O);

4. Does the first token of one name match the second name? E.g. "Pepsi Cola" equals "Pepsi". This is case-independent;

5. Is one of the names an acronym of the other? E.g. "ICI" is equivalent to "Imperial Chemical Industries". This check is also made disregarding an initial "The" or trailing company designator, e.g. "plc", "Co." or "Ltd." (O);

6. If one of the tokens in either of the names is one of a specified list of separators, e.g. "&", then if the token before the separator matches the other name, then they are equivalent. "Macy" matches "R.H. Macy & Co." or "Frank Loyd de Paris" matches "Loyd" (O,P);

7. Do the names match after stripping off "The" and/or a trailing company designator (if one of these is present)? E.g. "The Magic Tricks Co" is equivalent to "Magic Tricks" or "Magic Tricks Co" or "The Magic Tricks" (O);

8. Does one of the names match the token in the other which is immediately before a trailing company designator? E.g. "R.H. Macy Co" is equivalent to "Macy" (O);

9. Do the names match, except for case? E.g. "American Foods" is equivalent to "AMERICAN FOODS" (A);

10. Is one name a reversal of the other, reversing around prepositions or determiners only? E.g. "Defence Department" equals "U.S. Department of Defence" (O).

11. Does one name consist of concatenated contractions of the first two tokens in the other name? E.g. "Communications Satellite" equals "ComSat", or "Pan American" equals "Pan Am" (O).

12. Do the first and last tokens in a multi-word name match the first and last tokens in the other? (P)

9.3 Limitations

The NAMEMATCH module is domain dependent to the extent that some of its rules are specific to organisation names. The rules are also language dependent, although many will be applicable across a number of languages.

Chapter 10

Discourse Interpreter

10.1 Overview

The DISCOURSE INTERPRETER module translates the semantic representation produced by the parser into a representation of instances, their ontological classes and their attributes, in the XI knowledge representation language (see [6]). XI allows a straightforward definition of cross-classification hierarchies, the association of arbitrary attributes with classes or instances, and a simple mechanism to inherit attributes from classes or instances higher in the hierarchy.

10.2 External Processes

10.2.1 Software Requirements

The discourse interpreter is written in Prolog and requires SICStus Prolog 3#6 or SWI Prolog 2.9. If SICStus is not found, the overall GATE build may fail to automatically create the `disint` executable Prolog saved state and the `mkdisint_sicstus` or `mkdisint_swi` script must be run manually, after specifying the path to the appropriate Prolog executable.

10.2.2 Input

The file of semantic representations given as input to the discourse interpreter should have an initial `doc_descriptor/1` term which specifies the document number to be used in all output, followed by a series of `semantics/4` terms, one for each sentence in the text, with the following format:

```
semantics(sentence_n:N1, section_n:N2, type:Type, Semantics).
```

`Semantics` is a Prolog readable list of the semantic representations derived from sentence N1. The semantic representations are as described in Chapter 8. The `Type` specifies whether the sentence N1 is part of the header, body or trailer of a text.

The following is a complete well-formed input:

```
doc_descriptor('940224-0133').
semantics(sentence_n:1, section_n:3, type:body, [
  person(e1),
  name(e1,'James'),
```

```
title(e1,'Mr.'),
be(e2),
tense(e2,present),
title(e3,'president'),
lsubj(e2,e1),
lobj(e2,e3) ]).
```

A `name_match` attribute is added to the semantic representation from the parser to specify a list of instances with compatible names, as identified by the NAMEMATCH module (see Chapter 9), e.g. `name_match(e1,[e4,e5,e6])`.

10.2.3 Output

The discourse interpreter writes out a Prolog readable version of the final discourse model, using XI notation for class definitions, and `props/2` terms for lists of attributes. Two special attributes, `muc_ne` and `muc_coref`, are added to flag instances which should be included in MUC NE and CO SGML results.

Coreference output

Coreference flags are added to all instances of the `object` class with more than one `realisation` attribute specifying a byte offset pair. This will occur as a result of merging two instances from the original input and will therefore represent what the system considers to be a single instance with multiple realisations in the text.

Named Entity output

All instances of the MUC NE classes `organisation, location, person, money, percent, date` and `time` are assigned a `muc_ne` and an `ne_tag` attribute. The `ne_tag` attribute specifies a byte offset range in addition to the `realisation` attribute, both to record named entity offset ranges after any coreference merging, and also to allow ranges within the realisation range, such as a person name excluding its title, to be marked up separately, as required for the MUC NE task.

10.2.4 Resources

The World Model

The definition of a XI cross-classification hierarchy is referred to as an *ontology*, and this together with an association of attributes with nodes in the ontology forms a *world model* (WM). Processing a text acts to populate this initially bare world model with the various instances and relations mentioned in the text, converting it into a world model specific to the particular text, i.e. a *discourse model* (DM).

The attributes associated with nodes in the ontology are simple `attribute:value` pairs where the value may either be fixed, as in the attribute `animate:yes` which is associated with the `person` node, or where the value may be dependent on various conditions, the evaluation of which makes reference to other information in the model. Certain special attribute types, `presupposition`, `distinct` and `consequence`, may return values which are used at particular points during processing to modify the current state of the model, as described in the following

section. The set of attribute-value structures associated with the whole ontology is referred to as an *attribute knowledge base*.

The files comprising the world model are assumed to reside in a `.../creole/disint/wm` directory, which may be a link to a directory for a particular domain. A top level `wm.pl` file is loaded from this directory, which then loads all files containing declarations for the world model. This file is read at compile time only, and a single `static_wm.pl` file is then dumped out, containing compiled forms of all the world model declarations together.

As a convention, the file `ontology.pl` contains only XI class definitions, `generic_kb.pl` contains general purpose attributes, and `scenario_kb.pl` contains attributes specific to the current task. Each of these files may be further split, e.g. `coref_kb.pl` in the MUC-7 world model contains all `distinct` attributes from `generic_kb.pl`.

XI compilation

The compiled form of the world model declarations, as dumped out to the file `static_wm.pl` at compile time and used internally during runtime processing, is intended to rewrite the XI notation in a more efficiently accessible form in the Prolog database. It also represents certain inherited relationships directly, to attempt to avoid repeated expensive processing at runtime.

Facts added to the discourse model at runtime are not compiled, and so all processing allows for the use of compiled and uncompiled forms simultaneously (possibly at the expense of some of the efficiency improvements gained by compiling). Facts obtained from the static model and from the text can therefore always be distinguished during processing.

The discourse model written out on completion of the module includes only the uncompiled facts obtained from the text. The compiled facts are assumed to remain fixed and always available in the `static_wm.pl` file if required, as in the Pixi discourse model viewer or the Template Writer module.

NB: Attributes are only compiled for nodes or instances which have also been compiled. A `props/2` declaration for a class or instance which does not have its type defined elsewhere will therefore be ignored.

10.2.5 Parameters

The discourse interpreter can be run either from the command line, as it is called by the GATE wrapper, or in interactive Prolog mode useful for tracing and debugging. Both modes take the following options:

-v write progress messages to standard output.

-d write debugging information to standard output.

-o `filename` redirect standard output to a file.

-s `filename` input file of semantic representations (compulsory).

Input files for particular texts can be obtained by running the discourse interpreter from the GGI interface, then interrupting the Prolog process and copying the most recent file with an `_dump` suffix from the `/tmp` directory.

A final (compulsory) argument is assumed to be a Prolog goal to run one of the discourse interpreter's substages: `add_semantics`, `add_presuppositions`, `object_coref`, `add_consequences` or `event_coref`, or `disint_all` which runs all stages together.

The interactive Prolog mode, for SICStus, is started as follows (`%` indicating the shell prompt, and `?-` indicating the Prolog prompt). The `-s` flag and input file, together with the Prolog goal to run are the only compulsory arguments.

```
% sicstus -l mkdisint_sicstus.pl
?- disint(['-v','-o',output_file,'-s',input_file,disint_all]).
```

10.3 Processing

Figure 10.1 shows the main components within the Discourse Interpreter.

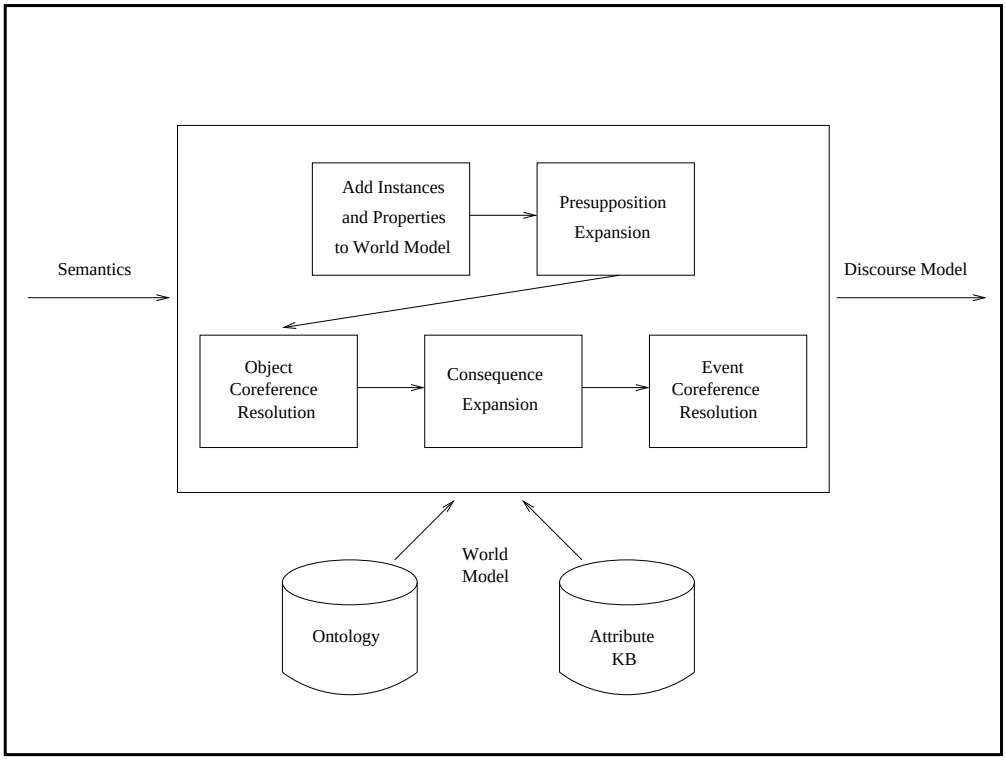


Figure 10.1: Discourse Interpretation

Each of the five stages of discourse interpretation is repeated for each sentence in the input. For newswire applications the header is processed last, to reduce spurious coreference into the header which poses difficulties of analysis because of its telegraphic style and initial letter, or indeed complete, capitalisation of most words.

10.3.1 Add Semantics

The semantic representation produced by the parser for a single sentence is processed by adding its instances, together with their attributes, to the discourse model which has been

constructed so far for the text. Instances which have their semantic class specified in the input (via unary predicates) are added directly to the discourse model beneath their class node, if the class node already exists in the ontological hierarchy. The class node is presumed to be labelled with the instance's predicate (so, e.g. given `company(e1)` instance, `e1` is added beneath the class node labelled `company` in the discourse model) or retrievable via a language-specific `concept_index`. A set of `concept_index/4` facts can be used to define a mapping between root forms in the input and nodes in the discourse model. For example `concept_index(firm, en, n, company)` would map the unary predicate `firm`, when derived from input containing the English word *firm* in its nominal form, to the node labelled `company` in the discourse model. This allows for a simple treatment of synonomy and also allows the same world model to be used with input texts from different languages. So, if a French parser can deliver semantic representations in the form we have been discussing, only with French root forms as predicates in the representation language, then they may be mapped to nodes in the discourse model which bear labels in another language, natural or artificial. E.g. `concept_index(compagnie, fr, n, company)` will map occurrences of `compagnie` derived from input containing the French word *compagnie* in nominal form to the discourse model node labelled `company`. This permits the reuse of domain models across source languages.

If the class specified in the input does not exist in the hierarchy and cannot be mapped by the index (say, `penguin(e23)`), a new class node (`penguin(_)`) is created dynamically under either the `event(_)` or `object(_)` nodes, with `event` instances in the input being distinguished from `object` instances by the presence of event-like attributes, i.e. `time`, `lsubj` or `lobj`. This facility allows a crude, high level categorisation of unknown classes.

Attributes – binary predicates in the input in which the first argument is always an instance identifier – are added to the attribute-value structures associated with all the instance identifiers occurring within them. So, e.g., `lobj(e1,e2)` will be added to the attributes associated with `e1` and to those associated with `e2`.

Sentences from the text body are added to the discourse model first, with any header processed after the rest of the text. This is done mainly to assist the classification of proper names, which are commonly only used in abbreviated form in headers, but usually in full form in the text body. The full form allows a more reliable classification, against which terms in the header can then be matched via the coreference mechanism. Only proper nouns are allowed to refer from the header to the body, with pronouns and common nouns being restricted to corefer within the header.

Text Structure

As instances are added to the discourse model, a special 'text' subhierarchy is constructed to record the structure of the text and which instances were introduced where. A hierarchy of header/body and numbered section and sentence nodes is created dynamically in the model, below a pre-existing `text(_)` node. The hierarchy is created using the values of the `sentence_n`, `section_n` and `text` features included in the `semantics/4` terms in the input. Sentences are represented as instances (e.g. `sentence1`) below section nodes (e.g. `section0(_)`), and an attribute of each sentence instance records a list of all instances mentioned in that sentence, preserving the order in which they were introduced. Inherited attributes of the `text(_)` node can then be used to return lists of specific types of instances from each sentence, section, body, or indeed the entire text, i.e. `events`, `objects`, `proper_nouns`, `pronouns` or `common_nouns`.

10.3.2 Add Presuppositions

Following the addition of the semantics for each sentence, the model is checked for any inheritable `presupposition` attributes of the instances or attributes just added. Any values returned are used to add (or remove) further information in the model. I.e. attributes of the form `presupposition(X,[P1,...,PN]) :- C` associated with a node `X` in the attribute knowledge base (see 10.2.4) will cause the predicates `P1, ..., PN` to be added to the discourse model, should any associated conditions `C` be met. Any variables in `P1, ..., PN` which have not been bound to instance ids during the inheritance of the the `presupposition` attribute will be bound to new instances during the process of adding the predicates to the discourse model (i.e. new individuals will be hypothesised). Finally, if the functor of `Pi` is `xi_call`, then its argument will be executed rather than `Pi` being added to the discourse model. This allows an arbitrary procedure to be called as a side-effect of adding something to the discourse model. At present this is the only way to cause instances or properties to be removed from the discourse model, as a consequence of something else being added.

This presupposition mechanism has a variety of uses including:

1. *Inferring Semantic Class Information* Missing semantic class information for instances in the input semantic representation may be derived from type restrictions on attribute arguments. For instance, a `presupposition` attribute associated with the node in the ontology corresponding to the `name` attribute, records that this attribute holds only of entities of type `object`. This is expressed in the `generic_kb.pl` file (see 10.2.4) as follows:

   ```
   props(name,[
        (presupposition(name(E,Name), [object(E)]) :-
            \+(E <- object(_)))
   ]).
   ```

   Note that the `presupposition`'s holding here is conditional on `E` not already being an instance of type `object`. Thus, when attempting to add say `name(e3,Jones)` to the model, then in the absence of any more specific information about the type of `e3`, such as `person(e3)` in the input, `e3` will be added as an `object`. That is, the default semantic type of named entities is `object`, as opposed to, say, `event`.

2. *Entity Hypothesis* Expected, or implicit, instances, can be hypothesised to be resolved by the coreference mechanism. Nominalisations of verbs can be identified by presuppositions and lead to the hypothesis of the corresponding event, for example a hypothesised `launch_event` from an instance of a `launch` object. This facility is particularly useful in the context of scenario-based information extraction tasks (such as the MUC-7 rocket launch scenario) where nominalisations of the relevant scenario events form a significant proportion of the events to be captured. Similarly, instances of indirectly related scenario specific objects, such as `mission`, can also give rise to the hypothesis of a `launch_event`.

3. *Word Sense Disambiguation* Instances whose semantic classes have been derived from word roots with multiple senses can be examined and potentially reclassified if their semantic and syntactic context is at odds with the primary sense stored in the world

model. For example, the MUC ontologies contain `fall` as a subclass of `date`, but an instance of `fall` initially added here may be removed by a presupposition that identifies a particular instance as, say, referring to a fall in share prices rather than a date. Scenario-specific senses can also be caught in this way, for example only `fire_event` instances related to missiles are retained as potential `launch_event`s in the MUC-7 scenario, and instances pertaining to, say, dismissal events are reclassified.

4. *Role Classification* Role information is frequently conveyed via nominalisations in natural language (*lawyer*, *mother*), but in the underlying representation it may be appropriate to store this via attributes on more basic types. This can be achieved by creating an ontology with a hierarchy of person roles, for example job roles and family roles, with cross classification of instances permitted. A general presupposition on the `person_role` node can then act to reclassify instances of these nodes from the text as instances of the `person` node, with a property indicating the role. This avoids the requirement to specify all person roles as subclasses of person to force semantic compatibility for coreference, so that, e.g., one and the same person can be both a lawyer and a mother.

5. *Partial Parse Extension* Extension may be attempted of partial parses, identified by the absence of compulsory attributes for certain instance types, mainly events. For example, a `presupposition` attribute of all event nodes causes the introduction of a new instance for event classes which require a logical subject (currently all event classes), if there is no explicit `lsubj` attribute in the input. The new instance will have a `lsubj` attribute linking it to the event instance, and it will be flagged as a 'hypothesised' instance and assigned an identifier of the form `a1` to distinguish it from instances explicitly mentioned in the input representation of the text. The hypothesised instance may also be of a semantic type required by a specific event type. So, for example the following expresses the requirement that should no subject already be known (perhaps because of parse failure or ellipsis) for a verb belonging to the class of `launch_event`, then an instance of type `organization` should be hypothesised in the relation of `lsubj` to the `launch_event`.

```
props(launch_event(E),[
     (presupposition(E, [organization(X),lsubj(E,X)]) :-
         \+(nodeprop(E,lsubj(E,_))))
]).
```

Following the addition of all presuppositions of instances from the current sentence, the coreference mechanism is called to attempt to resolve all hypothesised instances with other instances in that sentence. The coreference mechanism operates as described below, but with a restricted set of potential antecedents, and including a special set of `distinct` attributes, specifying coreference restrictions which only apply to hypothesised instances. These include restrictions on the ordering of subject and verb, stating that the antecedent for a hypothesised subject must occur before the verb expressing the event in an active construction, or after if passive. The relative positions are established by checking the `realisation` attributes, using only the most recent if there are multiple values, to ensure that the realisation used is from the current sentence and not from the result of some previous coreference. Any hypothesised instances remaining unresolved after coreference has been attempted are removed entirely from the discourse model

Presuppositions can also be used to attempt prepositional attachment, where the parser has left a prepositional phrase unattached. The parser typically generates semantic representations of the form, e.g. `on(e1,e2)`, where the preposition itself becomes the predicate, the first argument is the entity to which the phrase attaches, and the second argument is the complement of the preposition. For an unattached prepositional phrase the `e1` argument will figure in no other term in the semantic representation. In this case presuppositions which use information including which preposition is involved, the semantic type of its complement and the semantic types of other instances in the sentence can be used to decide with which instance `e1` should be merged.

10.3.3 Object Coreference

Following the addition of the instances mentioned in the current sentence, together with any presuppositions that they inherit, the coreference algorithm is applied to attempt to resolve, or in fact merge, each of the newly added instances with instances previously in the discourse model. Coreference resolution is performed by comparing the following sets of instances in this order.

1. compare: each instance mentioned in the current sentence using a proper noun
   with: every other instance in the discourse model which was mentioned using a proper noun

2. compare: each instance mentioned in the current sentence
   with: every instance before it in the current sentence

3. compare: each instance mentioned in the current sentence using a pronoun
   with: every instance mentioned in the current paragraph[1]

4. compare: each instance mentioned in the current sentence using a 'normal' noun (i.e. not a proper noun or pronoun)
   with: every instance mentioned in the current or previous paragraphs

These comparison sets effectively embody distance restrictions on the potential coreferences of the various noun types: proper nouns have no distance restriction, pronouns can only refer within the same paragraph (but see footnote), and normal nouns can only refer within two paragraphs. This last restriction on normal noun coreference was introduced mainly for reasons of efficiency, limiting the size of the comparison set when processing large texts.

Each comparison set may be viewed as a set of *candidate sets*, a candidate set being a set of pairs of instances all of whose first elements are the same (an instance in the current input) and whose second elements are possible instances, or candidates, occurring earlier in the text with which the first element might corefer. The algorithm proceeds as follows. For each pair of instances in each candidate set in each of the comparison sets listed above [2]:

[1] The previous paragraph in the case of an initial pronoun if the current sentence starts a new paragraph. **NB:** This was changed for MUC-7 so that no distance restriction is applied to pronouns not classed as pleonsatic. Successively earlier sentences are considered until a compatible antecedent is found.

[2] While the order in which instance pairs within a candidate set are examined cannot affect outcome of the algorithm, the order in which the candidate sets of a given comparison set are processed may indeed do so. We have not yet done any testing to determine just how significant this effect may be.

1. Ensure semantic type consistency
   The semantic types of the two instances must be ordered in the ontology. If this is true a semantic similarity score is calculated using the inverse of the length of the path (measured in nodes) between the two classes. The attempt to resolve the two instances is abandoned if the semantic types are not ordered. For example, `person` and `company` are not ordered with respect to each other in the ontology and therefore no pair of `company` and `person` instances would ever be coreferred. An instance of type `company` could be coreferred with one of type `organisation` or with one of type `object`; other things being equal, the former pair would be preferred on the grounds of higher semantic similarity.

2. Ensure non-distinctness
   Any additional coreference constraints are checked at this point to ensure that the pair of instances currently being considered do not possess any characteristics which imply that they should not be resolved. For instance, one of the constraints specifies that a new instance which has been introduced by an indefinite noun phrase in the text, should not be permitted to refer to any existing instance. The constraints are represented via the `distinct` attribute of certain nodes in the ontology, and should the current pair of instances inherit this attribute, the attempt to resolve them is abandoned. The various constraints currently implemented are discussed in more detail below.

3. Ensure attribute consistency
   The values of any fixed single-valued attributes (as classified in the ontology, e.g. `animate`) common to both instances, must be identical. The attempted resolution is abandoned if any conflict is found.

4. Calculate a similarity score
   The semantic similarity score is summed with an attribute similarity score to give an overall score for the current pair of instances. The attribute similarity score is established by finding the ratio of the number of shared multi-valued attributes with compatible values, against the total number of the instances' attributes. Certain attributes, notable `name`, are weighted strongly to boost the similarity score if the values are shared (or, in the case of the `name` attribute, compatible using the value of the `name_match` attribute).

After each pair in a candidate set of a comparison set has either been assigned a similarity score or has been rejected on grounds of inconsistency, the highest scoring pair of instances in the set (if any score at all) are merged in the discourse model. If several pairs have equal similarity scores then the pair with the closest realisations in the text (actually, the most recently compared) is preferred.

The merging of instances involves the removal of the least specific instance (i.e. the highest in the ontology) and the addition of all its attributes to the other instance. This will result in a single instance with more than one `realisation` attribute, which corresponds to a single entity mentioned more than once in the text, i.e. a coreference as required by the MUC task.

On each merging of instances, the attributes of all instances mentioned in relational attributes of the instance being removed are checked and updated. This assumes that all relational attributes (i.e. attributes where both arguments of the binary predicate are instance identifiers) occur in the attribute lists of both the instances mentioned. Without this

restriction, the entire discourse model would need to be searched to ensure all occurences of a removed instance are updated.

Note that this whole process of establishing coreference followed by merging instance ids is deterministic: once two instances have been merged no subsequent evidence can cause them to be unmerged (this constrasts with some treatments of coreference in which coreferences are represented by equations between instance ids and attributes determined by equational reasoning; such an approach allows equations to be retracted in the light of subsequent evidence, but at the cost of significantly greater computational complexity).

Additional Constraints

The constraints on coreference represented via the `distinct` attribute act to rule out the potential coreference of an instance pair which may otherwise be permitted by the base algorithm. The constraints used in LaSIE for the final MUC evaluation were established through training on the coreference data provided for the MUC dry-run evaluation.

The basic set of constraints as used in MUC-6 are as follows:

1. Prevent indefinite nouns from referring backwards
   A new instance introduced using an indefinite determiner is defined as being distinct from all other instances in the various comparison sets considered by the base algorithm, i.e. all instances before it in the text. For example, the phrase "an American company" would not be permitted to refer to any company previously mentioned in the text, reflecting an assumption that all indefinite determiners are used to introduce instances into a text for the first time.

2. Prevent non-pronouns from referring back to pronouns
   New instances mentioned using either proper nouns or full nouns are distinct from earlier instances which have been mentioned only by pronouns, i.e. preceding pronouns which could not be resolved with anything. An unresolved pronoun is thus prevented from being used as the root of a coreference chain in a text — roots must always be proper nouns or full nouns.[3]

3. Prevent unclassified proper names from referring back to dates
   New instances with `name` attributes but with a semantic class no more specific than `object` are defined as distinct from all instances with a semantic class of `date`. This reflects the assumption that the recognition of date proper names at the earlier stages of processing is complete and correct, and so an unclassified name must be of some other semantic type.[4]

4. Prevent non-proper nouns used as qualifiers from coreferring
   An instance introduced as a qualifier or modifier of another instance is distinct from all other instances. Thus, no instance is permitted to corefer with the instance of

[3]Several exceptions to this constraint are allowed: a noun which is the object of the verb *to say* can refer back to a first person pronoun, as in " '*I* agree', said *the chairman*". Clearly there will be generalisations of this case, but these should more properly be covered via a specific treatment of quoted speech, which is lacking in the current system. Pronouns within copular constructions can also refer ahead, as in "*This* is *a mystery*."

[4]LaSIE's performance on date expressions in the MUC named entity task has typically been around 95% combined precision and recall.

the class `video` mentioned in the phrase "the video manufacturers". Unfortunately this constraint rules out a class of coreferences which are explicitly included in the final MUC coreference task definition. However, on the dry-run evaluation data, the constraint produced a useful increase in precision and so was retained for this reason.

5. Prevent pronouns from referring back to dates, numbers or locations
   This constraint is probably the most domain specific of those used in LaSIE. An apparent feature of financial texts is that repeated references to particular instances of dates, numbers or locations are rarely made, especially pronominal references. They are therefore disallowed altogether in LaSIE. However, the adverbs *there* and *then*, which may more commonly refer to dates and locations, are not treated specially at present.

The above constraints, applied via the `distinct` attribute, are all associated with the `object` node in the ontology, and are therefore inherited by all instances considered by the base coreference algorithm. The following two constraints are associated with the `date` node only:

6. Prevent proper noun dates from referring back to non-proper noun dates
   A new instance classified as a date by the previous named entity recognition stages, is distinct from all preceding dates without proper names. The assumption here is that a date will not be introduced as a normal noun phrase, such as "the month", and then later referred to by its full form, e.g. "January".

7. Prevent non-proper noun dates without definite determiners from referring backwards
   All new instances of dates which were introduced using normal nouns with no definite determiner, are distinct from all preceding date instances. For example, the instances of the class `year` derived from the phrases "years ago" or "23 years old" would not be permitted to corefer with anything, whereas the constraint would not apply to the instance derived from the phrase "this financial year".

The use of the `distinct` attribute provides a mechanism by which a wide variety of coreference restrictions can be expressed. Those listed above were all that were used in LaSIE for the MUC-6 evaluation, but further development occurred for MUC-7 and has continued since. Since these `distinct` attributes are declaratively expressed in the `coref_kb.pl` file which is a part of the world model, the application developer is expected to add further domain-specific constraints in order to optimise coreference performance in specific domains.

One further general constraint is worth mentioning here – the identification of 'pleonastic' or 'non-referential' instances of the pronoun *it*, as proposed by Lappin and Leass [17]. Although the identification makes reference to purely syntactic and lexical information it can still be expressed via a `distinct` attribute of the `object` node in the ontology.

Lappin's and Leass' test for pleonastic pronouns involves the recognition of patterns such as "It is **Modaladj** that **S**", where **S** is a sentence complement and **Modaladj** is a member of a set of lexical items such as *possible*, *useful*, *important*, etc. Such syntactic patterns can be identified within the discourse model in LaSIE, due to the preservation of much predominantly syntactic information via instance attributes in the semantic representation. For example, the above syntactic structure would have a predicate-argument representation of the following form:

```
pronoun(e1,it),
be(e2), tense(e2,present), lsubj(e2,e1), lobj(e2,e3),
adj(e3,important)
```

where e3 is the event instance described by the (verbal) head of the complement **S**. This allows, to a certain degree, the reconstruction of the original syntactic form from the semantics, providing a mechanism by which syntactic constraints can be expressed in the world model. The identification of syntactic patterns is, however, very much dependent on the performance of the parser and the grammar, and, as yet, their limitations have not been fully established.

10.3.4 Add Consequences

Following object coreference resolution, the model is checked for any inheritable consequence attributes of the instances added from the current sentence (allowing for any changes in the instance identifiers caused by coreference resolution). The consequence attributes are similar to presupposition attributes, but with several differences in the way in which they are applied:

1. Conditions on consequence rules are tested after the coreference mechanism, and can therefore refer to information obtained from outside the current sentence via the merging of coreferential instances.

2. The coreference mechanism is applied to a hypothesised instances arising from a consequence attribute as soon as it is proposed, rather than collecting and attempting to resolve them all together, as is done for presuppositions.

3. Hypothesised instances from a consequence attribute are not cancelled if they are not resolved immediately, as are hypotheses from presuppositions attributes. They remain active during the processing of subsequent sentences and so may be resolved with instances mentioned much later in the text. Active hypotheses may be explicitly cancelled though, for example by the introduction of another hypothesis of the same type arising from another instance later in the text.

consequence attributes are typically used for task specific inferencing, attempting to establish values for attributes which correspond to template slots in the final template. Objects and events for each template structure, and attributes for each template slot, are usually added to the ontology to represent the required information, and then inference rules added to determine the attribute values whenever an object or event of the required type occurs in a text.

A consequence attribute may cause the introduction of new hypothesised instances, usually with certain relational attributes to link it to instances from the text or from other parts of the template. The coreference mechanism is then called in two stages for each hypothesised instance, firstly to attempt to resolve the instance within the current sentence, and then, if permitted by checking the inherited value of an intersentential attribute, to attempt to resolve the instance across the entire text.

Before hypothesising a new instance, each consequence rule typically checks whether a value for the slot it is attempting to fill has already been established. Because the hypotheses of any previous rules will have already been resolved when the rule is called, this effectively gives a priority ordering of consequence attributes in the knowledge base files, with the more specific rules at a particular node occurring first, and general default rules at the end.

10.3.5 Event Coreference

Following the addition of all consequences, a task specific merging of event instances is attempted. Relevant event types are defined for each task with comparison routines specific to those types. Event coreference in LaSIE is better seen as partial template merging rather than a general mechanism to resolve anaphoric references to events. In general, templates are based around descriptions of particular event types, and several descriptions in a text may contribute to a single template, with each supplying additional information.

Following the addition of an event instance of a relevant type, and the attempted resolution of all hypotheses arising from it, the instance is compared with all other event instances of the same type throughout the text. If no conflicting attribute values are found, as defined by a comparison routine for each event type, and possibly additional `distinct` attributes, two event instances will be merged and then used to represent a single template. See [14] for further details.

10.4 GATE Interface

10.4.1 Input

`semantics` annotations, with a `qlf` attribute.
`name` annotations, with `type` and `matches` attributes.
`sentence` annotations.
`section` annotations.
`header` annotation - one only.

10.4.2 Output

`name` annotations, with a `type` attribute and an `sgml` attribute specifying the type in MUC format (and which is used by the GGI Export SGML function). The annotations completely replace the previous set of `name` annotations.
`coreference` annotations, with multiple spans.
`muc_coref` annotations, with single spans, representing coreference relations via an `sgml` attribute which specifies the relations in MUC format (and which is used by the GGI Export SGML function).

`xi_instance` annotations, with `class` and `props` attributes.
`xi_node` annotations, with `class` attributes.

Both the `xi_instance` and `xi_node` annotations may have multiple spans.

Each `xi_instance` annotation represents a single instance from the final discourse model, with its semantic class, in XI notation, and a list of its non-inherited attributes. The annotation will have a span for each of the instance's realisations in the text. Hypothesised instances in the final model, which have no realisation, are arbitrarily assigned a span covering the whole text.

The `xi_node` annotations each represent additional semantic classes added dynamically to the original 'static' world model during processing of the text. The value of the `class` attribute specifies the parent class in XI notation. The annotation spans, which may be multiple, are those of the instances which caused the introduction of the new class.

10.4.3 Parameters

Verbose writes progress information to the file `disint.log` in the current working directory.

Debugging writes additional debugging information to the file `disint.log`.

The log file can be viewed via the module's results menu, but on large texts the contents may be larger than can be displayed in the tcl text viewer, and the file must then be viewed outside the GGI.

10.4.4 Processing

The values of the `semantics` annotations within each `sentence` annotation span are concatenated and a `semantics/4` term constructed, as described above. The `section` and `header` annotations are referred to to determine the position of the current sentence and complete the term.

The `name_match/2` terms are produced directly from the `matches` attribute of the `name` annotations.

10.5 Limitations

Issues of synonomy (multiple surface forms mapping to the same 'concept' node) and word sense disambiguation (the same surface form mapping to distinct concept nodes) are largely ignored, though both can be addressed in this framework and have been in a limited way for particular applications. Predicates in the semantic representation are derived directly from the morphological roots of words, and the default behaviour is to map directly from them to concept nodes in the discourse model labelled by these word names. Synonomy can be dealt with in a rudimentary way, as described in 10.3.1 above, by use of a `concept_index` mapping table which allows multiple surface forms to be mapped to a single concept node. Word sense disambiguation (discussed in 10.3.2 above), for example the problem of mapping surface references to "a river bank" and "a financial bank" to different concept nodes in the discourse model, can be addressed via `presupposition` attributes of a generic `bank` node which re-attach instances of type `bank` to the appropriate concept node (`river_bank` or `financial_bank`) by checking available syntactic/semantic context. Clearly this approach is cumbersome, as it requires rules to be written for each ambiguous word. Nevertheless, it has not led to serious problems in information extraction applications (perhaps because of the 'one sense per discourse' phenomenon [11]). A preferable solution might be to introduce a word sense disambiguation module into the system and use, e.g., word root plus sense number as predicates in the semantic representation language and as node labels in the world/discourse model.

Chapter 11

Template Writer

11.1 Overview

The TEMPLATE WRITER module reads the final discourse model produced by the discourse interpreter, selects those instances which represent template structures with values for all the required attributes, then formats and writes out the values. Three separate results are produced, written to separate files, for the MUC Template Element, Template Relation and Scenario Template tasks.

11.2 External Processes

11.2.1 Software Requirements

The template writer is written in Prolog and requires SICStus Prolog 3#6 or SWI Prolog 2.9. If SICStus is not found, the overall GATE build may fail to automatically create the `template` executable Prolog saved state and the `mktemplate_sicstus` or `mktemplate_swi` script must be run manually, after specifying the path to the appropriate Prolog executable.

11.2.2 Input

A Prolog readable file containing a `doc_descriptor/1` term specifying the WSJ or NYT document number, or the text's filename for other text types, a `gensymmark/1` term specifying the last hypothesised instance identifier created by the discourse interpreter, and a series of XI `<--/2` instance type declarations, together with `props/2` attribute lists for each instance. There may also be XI `==>/2` semantic class declarations for classes added dynamically during discourse interpretation.

The module also reads in, at compile time, the `static_wm.pl` file produced by the discourse interpreter at compile time. This file is assumed to exist in the `.../creole/template` directory as a link to the appropriate `.../creole/disint/wm` directory. Several other files are also assumed to be shared with the discourse interpreter, in particular `kb_utils.pl`, though these are not modified by the compilation of the discourse interpreter. Any changes to the world model will therefore require the recompilation of both the `disint` and `template` executable Prolog saved states, in that order.

11.2.3 Output

Files specified by the parameters described below, with formats as described by the BNF definitions in the MUC task specification documents.

11.2.4 Parameters

The template writer can be run either from the command line, as it is called by the GATE wrapper, or in interactive Prolog mode useful for tracing and debugging. Both modes take the following options:

`-t filename` write Template Element results to a file.

`-r filename` write Template Relation results to a file.

`-s filename` write Scenario Template results to a file.

A final argument given to the template writer is assumed to be the input discourse model file. Input files for particular texts can be obtained by running the template writer from the GGI interface, then interrupting the Prolog process and copying the most recent file with an `_dump` suffix from the `/tmp` directory.

The interactive Prolog mode, for SICStus, is started as follows (`%` indicating the shell prompt, and `?-` indicating the Prolog prompt). The input file and any one of the parameters are the only compulsory arguments.

```
% sicstus -l mktemplate_sicstus.pl
?- template(['-t',te_out,'-r',tr_out,'-s',st_out,input_file]).
```

11.2.5 Processing

The input file is compiled into the `template` executable Prolog saved state at run time, and, combined with the `static_wm.pl` read in at compile time, reproduces the final discourse model from the discourse interpreter. The split between the discourse interpreter and template writer modules is artificial, because the template writer results could be produced directly from within the discourse interpreter. The separation is to allow the substitution of alternative template writers, each producing different results from a single discourse model.

Template Element output

The discourse model is searched for all instances of classes required in the output, i.e. `organisation`, `person` and `artifact` for MUC. A table of all possible slots is then built for each instance, and values searched for from the corresponding, possibly inherited, attributes, which may refer to attributes of other related instances. Tables which include values for at least the `name` or `descriptor` slots are then formatted as required by the MUC task definition, and written to the output file.

A unique identifier is required for each template element, and is made up of the `doc_descriptor` and the identifier of the corresponding instance in the discourse model. The template element numbers are therefore not sequential, but can be used to map back to the discourse model directly.

Template Relation output

This proceeds as for template elements, but the initial set of instances is found by searching the discourse model for all instances with particular attributes, for MUC either `location_of`, `employee_of` or `product_of`. A template relation entry is generated for each attribute found, and simply numbered sequentially. All instances mentioned in template relations are then also written out as template elements to the same file.

Scenario Template output

For MUC-6, all instances of the top level `succession_event` template class are retrieved from the discourse model, then for each one an attempt is made to establish values for a related organisation and post, and, via associated `in_and_out` instances, at least one person. Output is only produced for events with values for all these attributes.

Multiple values for the `post` attribute are permitted for a single event, resulting from conjoined or apposed posts in the text, and so each event instance may produce multiple `SUCCESSION_EVENT` entries in the output template. The numbering used in the output is based directly on the instance numbers in the discourse model, with the exception of output generated as a result of multiple posts. In this case new template numbers are generated incrementally using the `gensymmark/1` value from the input file, to avoid any duplication. Template entries for the instances related to each `succession_event` are then written out as template elements to the same file.

For MUC-7, the top level template class is `launch_event`, which requires related `payload` and `vehicle` objects. See the MUC task specification document for full details of the obligatory slot values, each of which is checked before producing any output.

11.3 GATE Interface

11.3.1 Input

`xi_instance` annotations, with `class` and `props` attributes.
`xi_node` annotations, with `class` attributes.

The template writer wrapper calls the `creole_disint_dump_dm` routine defined in the discourse interpreter wrapper to read off the required annotations and write the Prolog format input file.

11.3.2 Output

Output files are written to the directory from which GATE was started. The filenames use the original input text filename with `_te_dump`, `_tr_dump` or `_st_dump` appended as appropriate. The contents of the output files are then read in and stored as the values of:
`template_elements` document attribute,
`template_relations` document attribute,
`scenario_template` document attribute.

11.4 Limitations

The template writer is highly domain and task dependent. Any change to the template definition requires the source code to be modified.

Chapter 12

MUC Scorer

12.1 Overview

The MUC scorer was supplied by the MUC organisers, but has been integrated into LaSIE to allow results to be rapidly computed and viewed during IE application development. The MUC scorer is a single program which, appropriately parameterised, can compute precision and recall results for each of the five MUC-7 tasks: Named Entity, Coreference, Template Element, Template Relation, and Scenario Template. To allow these to be run and viewed independently the scorer has been integrated as five separate CREOLE modules, one per task.

The scorer can be used for templates other than those defined for the MUC-7 tasks, provided the syntax of the output templates or SGML annotations (in the case of the Named Entity Task) conforms to that expected by the scorer. To do so requires modification of the configuration files supplied with the scorer. Of course, using the scorer for any task presupposes the existence of answer key files against which the output of the system is to be measured.

12.2 External Processes

12.2.1 Software Requirements

The MUC scorer is a C program, Copyright 1998, Science Applications International Corporation. Source code and binaries were made freely available to MUC participants, but interested parties should now contact SAIC directly. See `www.muc.saic.com` for further information and full documentation.

12.3 GATE Interface

12.3.1 Input

`name` annotations, with `sgml` attributes, for the NE results.
`muc_coref` annotations, with `sgml` attributes, for the CO results.

`template_elements` document attribute, for the TE results.

`template_relations` document attribute, for the TR results.
`scenario_template` document attribute, for the ST results.

12.3.2 Output

`score_report` document attribute.
`report_summary` document attribute.

The files `scores` and `report_summary` are also left in the directory from which GATE was started.

12.3.3 Parameters

Config The filename of the configuration file for the scorer. A modified version of this will be written to the directory from which GATE was started, as `.muc_config`, for use when the scorer is run.

Keys The directory in which key files for the scorer can be found. Key files are assumed to have the same filename as the text being scored with `_ne_key`, `_co_key`, `_te_key`, `_tr_key` or `_st_key` as a suffix.

Bibliography

[1] S. Azzam, K. Humphreys, and R. Gaizauskas. Using coreference chains for text summarization. In *Proceedings of the ACL'99 Workshop on Coreference and its Applications*, Baltimore, 1999.

[2] E. Brill. A simple rule-based part-of-speech tagger. In *Proceeding of the Third Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy, 1992.

[3] H. Cunningham, K. Humphreys, R. Gaizauskas, and M. Stower. *CREOLE Developer's Manual.* Department of Computer Science, University of Sheffield., 1996. Available at `http://www.dcs.shef.ac.uk/research/groups/nlp/gate`.

[4] Defense Advanced Research Projects Agency. *Proceedings of the Sixth Message Understanding Conference (MUC-6).* Morgan Kaufmann, 1995.

[5] Defense Advanced Research Projects Agency. *Proceedings of the Seventh Message Understanding Conference (MUC-7)*, 1998. Available at `http://www.saic.com`.

[6] R. Gaizauskas. XI: A Knowledge Representation Language Based on Cross-Classification and Inheritance. Technical Report CS-95-24, Department of Computer Science, University of Sheffield, 1995.

[7] R. Gaizauskas, L.J. Cahill, and R. Evans. Description of the Sussex system used for MUC-5. In *Proceedings of the Fifth Message Undersanding Conference (MUC-5)*, pages 321–335. ARPA, Morgan Kaufmann, 1993.

[8] R. Gaizauskas and K. Humphreys. A combined ir/nlp approach to question answering against large text collection. In *Proceedings of RIAO 2000: Content-Based Multimedia Information Access*, pages 1288–1304, Paris, 2000.

[9] R. Gaizauskas, P. Rodgers, H. Cunningham, and K. Humphreys. *GATE User Guide.* Department of Computer Science, University of Sheffield., 1996. Available at `http://www.dcs.shef.ac.uk/research/groups/nlp/gate`.

[10] R. Gaizauskas, T. Wakao, K Humphreys, H. Cunningham, and Y. Wilks. Description of the LaSIE system as used for MUC-6. In MUC-6 [4], pages 207–220.

[11] W. A. Gale, Church K. W., and Yarowsky D. One sense per discourse. In *Proceedings of the DARPA Speech and Natural Language Workshop*, Harriman, New York, February 1992.

[12] G. Gazdar and C. Mellish. *Natural Language Processing in Prolog.* Addison-Wesley, Wokingham, 1989.

[13] K. Humphreys, G. Demetriou, and R. Gaizauskas. Bioinformatics applications of information extraction from journal articles. *Journal of Information Science*, 26(2):75–85, 2000.

[14] K. Humphreys, R. Gaizauskas, and S. Azzam. Event coreference for information extraction. In R. Mitkov and B. Boguraev, editors, *Proceedings of the Workshop on Operational Factors in Practical, Robust Anaphora Resolution for Unrestricted Texts*, pages 75–81, Somerset, NJ, July 1997. 35th Annual Meeting of the Association for Computational Linguistics, Association for Computational Linguistics.

[15] K. Humphreys, R. Gaizauskas, S. Azzam, C Huyck, B. Mitchell, H. Cunningham, and Y. Wilks. Description of the LaSIE-II system as used for MUC-7. In MUC-7 [5]. Available at `http://www.saic.com`.

[16] K. Humphreys, R. Gaizauskas, M. Hepple, and M. Sanderson. University of Sheffield TREC-8 Q & A System. In *Proceedings of the Eighth Text Retrieval Conference (TREC-8)*, 1999.

[17] S. Lappin and H.J. Leass. An algorithm for pronominal anaphora resolution. *Computational Linguistics*, 20:535–561, 1994.

[18] M.P. Marcus, B. Santorini, and M.A. Marcinkiewicz. Building a large annotated corpus of english: The Penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[19] G. A. Miller (Ed.). WordNet: An on-line lexical database. *International Journal of Lexicography*, 3(4):235–312, 1990.

[20] F.C.N Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis.* Number 10 in CLSI Lecture Notes. Stanford University, Stanford, CA, 1987.