Parmenides Technical Report TR-U4.3.1
CAFETIERE: Conceptual Annotations for Facts, Events, Terms, Individual Entities, and Relations

William J. Black, John M’Naught, Argyris Vasilakopoulos, Kalliopi Zervanou, Babis Theodoulidis and Fabio Rinaldi

January 11, 2005
## Contents

1. **The CAFETIERE formalism for basic semantic element extraction**  
   1.1 Basic semantic elements for text mining  
   2 Preprocessing  
   2.1 Document capture and zoning  
   2.2 Tokenization  
   2.3 Tagging  
   2.4 Gazetteer lookup  
      2.4.1 Limitations of current lookup module  
   3 Rule notation  
      3.1 Representation of text elements in rules  
         3.1.1 Features of phrases found by rule  
         3.1.2 Operators  
         3.1.3 Value expressions  
         3.1.4 Operator semantics  
      3.2 Exploiting the rule notation  
         3.2.1 A simple context free rule  
         3.2.2 Regular expression operators and ranges  
         3.2.3 Grouping of constituents  
         3.2.4 Pure grammar rules and semantic grammar rules  
         3.2.5 Variables and their uses  
         3.2.6 Illustrations of the use of features  
         3.2.7 Use of wildcards in string matching  
      3.3 The use of context  
      3.4 Co-reference between phrases  
         3.4.1 Potential of coreference  
         3.4.2 Coreference and apposition  
      3.5 Rules to create entity instances  
   4 Processing and Rule Order  
      4.1 Recursive rules  
      4.2 Repeatedly applicable rules  
      4.3 Phrase, entity, relation and event rule order
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Running the Analyser and Annotation Editor</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Adding annotations manually</td>
<td>25</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Entity annotations</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>Knowledge-base in place of gazetteer</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Role of the knowledge-base in analysis</td>
<td>31</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Step 1: Accessing properties of entities</td>
<td>32</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Step 2: Obtaining event constraints</td>
<td>34</td>
</tr>
<tr>
<td>6.2</td>
<td>Knowledge-base lookup</td>
<td>35</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Need for analysis-time access to the knowledge base</td>
<td>35</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The knowledge-base lookup mechanism</td>
<td>35</td>
</tr>
</tbody>
</table>
Chapter 1

The CAFETIERE formalism for basic semantic element extraction

The CAFETIERE\textsuperscript{1} formalism is a rule-based system for temporal text mining. The term “text mining” conjures up a range of activities in which large scale repositories of textual documents are filtered for content, using various techniques that go beyond mere indexing. Our own understanding of text mining (as participants in the Parmenides project) is one in which the “basic semantic elements” \cite{13} of texts are first extracted as references to facts/events, terms, individual entities and relations, and subsequently processed using data mining techniques to reveal additional relationships between objects and event instances. The preliminary phases of extraction are equally applicable to a range of concrete text processing applications, including indexing, term discovery and ontology extension, open-domain question answering and summarization.

1.1 Basic semantic elements for text mining

Among the basic semantic elements to be extracted from texts are: names of people, places, organizations, artifacts; terms designating unnamed objects and attributes of interest in a given domain, relations currently holding between objects, representations of events (of interest in a given domain), temporal attributes attaching to events, and temporal relations holding between events.

The standard approach to robust information extraction is to use a variety of antecedently existing processing and data resources, subjecting the text to each of several levels of analysis, and to correlate the attributes thereby assigned to text elements and spans with those of the items to be extracted. Proper name instances, for example, are identified and classified on the basis of a mixture of evidence, coming from different levels of analysis, e.g. orthography, lexicon, contextually disambiguated part of speech tag, gazetteers of whole names, part names such as common forenames and surnames, prefixes and suffixes to names such as styles and titles, etc. For example, the name “Maj. Gen. Walter C. Short” is easily recognizable (with some redundancy) as a person’s name, because of the military title abbreviations, Walter’s being a common English language first name, the orthography pattern \textit{capitalized, initial},

\footnote{The system described here has had different names in the past. The present name reflects the variety of information extraction and parsing sub-tasks that we claim it is suitable for.}
capitalized, and the fact that the part of speech tagger assigns NNP rather than NN or JJ as the tag for “short”.

The formalism (and supporting analysis engine) described in this paper is designed for this approach to robust text processing, where:

- a range of attributes characterize textual elements: the string, its orthography, its morphological root, its part of speech (or phrasal category if non-terminal), its semantic class or ontological category, linguistic attributes such as tense and aspect features, the arguments of the element as a predication, references to anaphoric antecedents, etc. Different subsets of these features are relevant to identifying and describing different sorts of text unit. This suggests that the ideal notation would allow selective reference to features where those that are irrelevant may be omitted.

- prior levels of analysis may characterize all text elements (as in part of speech tagging), or only selected elements (as in items found in a gazetteer). It can be useful to test whether a given feature exists for a text element.

- the identification of an element may depend on the prior identification of constituent elements. For example, the titles of many universities conform to the pattern “University of location”. For this pattern to be successfully matched in a text, the location would need to have been identified and labelled first. In general, the parsing or pattern-matching phase needs to be able to exploit matches already made. This suggests a pattern-matching mechanism in which newly discovered phrases can be built on, as in chart parsing, or as in a cascade of finite-state transducers.

- the identification of an element may depend on textual elements that it collocates with, not just with its constituent parts. For example, we could assume a sequence of capitalized words to name an author if they follow “by”, which is not itself part of the name. This suggest that the rule formalism should be context-sensitive.

- the identification of an element may depend on textual elements with which it is co-referent. For example, names are often given in full form on first mention, but in a shorter form subsequently. After the example in the previous paragraph, the surname only, “Short”, may be used, and it is the first mention that provides the main evidence that this refers to a person, and indeed to a particular person. Very often, this is dealt with by a separate component, but we find that some of the conditions under which text units co-refer can be expressed declaratively, in a simple extension to the rule notation.

- having identified an element, we need to be able to form a composite description from its attributes or component parts. Apart from being able to associate attributes with an individual entity (e.g. job-title and employer with a person), we also need to be able to represents facts or events. The ‘component parts’ of an event are the agent initiating it, entities affected by it, and circumstantial properties such as the time at which it occurred. Events may have complex components, as in an announcement, whose object is another event. This suggests that the rule notation should include the facility to build complex descriptions, or at least sets of feature-values, based on values found in constituent text units.

To meet these requirements, a declarative notation has been developed which assimilates characteristics found in unification-based and information-based linguistic formalisms [20]
with the facility for evaluating regular expressions over annotations (as in JAPE, [7, 14]), and with the ability to capture co-reference relations between phrasal elements. Other comparable mechanisms include those described by [18, 19].

The formalism has been supported with three different processing mechanisms:

1. a left-corner chart parsing algorithm, where any successful rule application may trigger the application of any rule, and one from competing analyses selected using a preference mechanism. This variant was developed for use in the FACILE project and is described in [2, 3, 6]

2. a cascade of finite state transducers (interpreted) in which the rules are applied strictly in sequence and newly labelled text spans replace their constituents. This version was developed for use in the CONCERTO project [4, 10], and is described in [5].

3. a compiled variant of 2.

More information about rule-invocation matters is given below, after discussions of processing prior to rule-based analysis and of the rule formalism.
Chapter 2

Preprocessing

Before rules are fired, several prerequisite analyses of the text are carried out. These are as follows:

2.1 Document capture and zoning

The document’s native markup is converted to the structural markup of the common annotation scheme [12, 15, 16, 17]. In doing this, front matter is separated from the body text, and the text is partitioned into paragraphs.

2.2 Tokenization

Within each paragraph, the text is partitioned into its basic elements: words, punctuation marks, numbers and special symbols. The text is subsequently seen as a single sequence of these basic text units. As each token is extracted, a token object is created, which as well as its string representation has attributes representing its position in the text, and an orthography code. This code is used in rules, and distinguishes lowercase, uppercase and capitalized words from each other and from other patterns such as multi-capitalized tokens which denote special coined symbols. The codes in use are discussed in table 3.2 below. However, as the codes in use may change, it is vital when designing or debugging rules to check the actual codes assigned to tokens. This can be done in GATE by the following sequence of actions:

1. Click the “Annotation sets” button.
2. Select the “Token” annotation set.
3. Click the “Annotations” button.
4. Click the “From” column heading (to order the annotations).
5. Repeatedly press the “page down” button on your keyboard or scroll with the mouse, until the annotation corresponding to the item in the text whose features you want to check appears.
2.3 Tagging

A tagger determines the part of speech of each token in the text. The current tagger has been trained using the transformation-based learning (Brill) algorithm, to induce a set of rules for contextually modifying initial tags supplied from a simple lexicon or by guessing. The tags in use, the tagset, are those of the Penn Treebank, listed in NLP texts such as [1] or [9].

The tagger’s accuracy may be a limiting factor in the performance of rule-based analysis. The ideal way to improve the tagger would be to derive a new initial lexicon from a corpus of pre-tagged text from the same domain and text-type (genre) as that to be processed. However, since the biggest contribution to tagging accuracy is the initial lexicon, users may instantly rectify tagging errors by adding the tokens tagged in error to the lexicon. The lexicon lists the tags of a word in frequency order. Tokens not previously in the lexicon can be associated with a tag with a frequency of 1.

The tagger lexicon is stored in the file `taglexfile.txt`, and is found in the directory `taggerdir`.

2.4 Gazetteer lookup

We label as semantic any classification of tokens according to their meaning in the field of the application. This could be done, on a broad coverage level, by reading information from a comprehensive resource such as WordNet about most content words. However, the practice in information extraction applications has been to make the processing application-specific by using lists of the semantic categories of only relevant words and phrases. Such a collection of lists or its representation in a database has been described as a gazetteer.

In the cafetiere system, the gazetteer is held in an Access or MySQL database with the data source name “lookup”. Currently, this database has one significant table, named `phrases` with attributes `firsttok`, `phrase` and `cat`. For a given word or phrase, there may be zero, one or more `cat` values, each in a separate database row. The field `firsttok` is simply the first word of each token, and using it initially as the key facilitates a simple algorithm for looking up multi-word phrases.

If a single-word token is found to have a match in the gazetteer, the `sem` value is added to the data structure representing the text unit’s features. If there are several `cat` entries for the same word, the `sem` feature’s value is a list representing the set of distinct `cat` values. If the match is of a phrase of more than one word, the phrase replaces its constituent parts in the token sequence.

Users can update the gazetteer directly by using the database’s facilities. To add the phrase “North Pole” as a location, the SQL query (2.1) could be used.

\[
\text{(2.1) } \text{INSERT INTO phrases VALUES ('north', 'north pole', 'location')} ;
\]

Alternatively, updates may be made through a graphical user interface. A Java program `DBUpdate` may be used for this purpose. It is not currently incorporated with the analyser.

2.4.1 Limitations of current lookup module

The most serious limitation of the current lookup module is that it does no more than associate a semantic category to a text span. Where different entities have the same semantic category,
the information from the gazetteer does not individuate them. For example, “Washington” as a city could designate Washington, DC or Washington, Tyne and Wear, UK. It would be preferable to retrieve a unique entity_id for each of the instances, as well as just a label assigning it to a class. This could be provided as an augmentation to the gazetteer by adding entity_id as a database column, and modifying the lookup code so that this additional information is returned.

However, since “Washington” could also designate Washington State and George Washington, we need to take care to pair up each entity_id with its own category. It would not be sufficient to say that the sem feature has the values \{city, province, politician\} and the id feature the values \{Washington_DC, Washington_UK, WA and George_Washington1\}, since the semantics of the rule-based analyser assume all such lists to be disjunctions, when in fact they are distributed disjunctions.

As it is planned to move to the use of a knowledge-base in place of the simple gazetteer, those kinds of enhancement will be delayed until that version of the resource becomes available (possibly as early as end January 2004). Section 6 describes the design of the knowledge-base lookup module.
Chapter 3

Rule notation

The rule notation has been specified as an EBNF grammar for processing by the AnTLR
compiler-compiler. It is substantially the same as the rule grammar of earlier versions, but
some minor improvements have been made to take advantage of the re-write. As a consequence
of using this declarative method of specifying the rule grammar, some kinds of enhancement
to the rule notation will be easy to implement.

A rule has one of the following forms:

(3.1) \( A => B\backslash C/D; \)

(3.2) \( A => B\backslash C/D\rangle\rangle E; \)

In both rule schemata, \( A \) represents the phrase that is recognized, and \( C \) represents the
text elements that are part of the phrase. \( B \) (which may be null) represents neighbouring
text elements that must be present immediately prior to \( C \), and \( D \) (also optional) represents
neighbouring text elements required to be present immediately following \( C \).

\( A \) is always instantiated by a single phrasal text element, and \( C \) by a sequence of 1 or
more text elements (the daughters of \( A \) in phrase structure grammar terms). \( \in \)–productions
are not permitted.

\( B \) and \( D \) are sequences of zero or more constituents. Many actual rules lack one or both
of such contextual constraints and therefore have one of the forms (3.3), (3.4) or (3.5). A
context-free rule has the form (3.5), in which it should be noted that the syntactic indicators
of the context boundaries (‘\( \backslash \)’ and ‘\( / \)’) are always required to be present.

(3.3) \( A => \backslash C/D; \)

(3.4) \( A => B\backslash C/; \)

(3.5) \( A => \backslash C/; \)

Within \( B, C \) and \( D \), text elements in sequences of > 1 are separated by commas, and may be
grouped by parentheses. Individual text elements or parenthesised groups may be followed
by regular expression operators or ranges.

In rule schema (3.2), \( E \) represents a text element already recognized anywhere in the left
context of \( C \), and is usually a co-referent (anaphoric) antecedent.
3.1 Representation of text elements in rules

An individual text element is described by a sequence of comma-delimited feature operator value expressions, enclosed in a pair of square brackets, as in (3.6).

(3.6) \[\text{syn}=\text{NNP}, \ SEM=\text{CITY}|\text{COUNTRY}|\text{PROVINCE}, \ orth!=\text{lowecase}\]

Features

like syn, sem\(^1\) and orth in (3.6) are atomic symbols, written as contiguous sequences of alphabetic characters (interpreted case-insensitively), underscores and hyphens. There is no restriction on the feature names that can be used, beyond the syntax just described. Table 3.1 lists the features may be used to access attributes ascribed to text elements by previous levels of analysis. The orthography codes are as listed in Table 3.2.

3.1.1 Features of phrases found by rule

On the left-hand side of a rule, any feature names can be used. What will be appropriate will vary from application to application. For the Parmenides project, the essential features associated with a rule-discovered phrase are its syntactic category and its semantic category. For these, the feature names syn and sem should normally be used. Note that syn used to be the same feature name as was used for the part of speech tag assigned to individual tokens in the PoS tagging phase of analysis. However, part of speech from the tagger is now labelled as pos. The token is assigned automatically to newly added phrases, and so also is the eid feature, which represents the entity identifier. In writing fact extraction rules, the eid feature should be used (rather than, say, the token) to identify the participants in the event.

3.1.2 Operators

Operators denote predicate functions applied to the attribute of the text element labelled by the feature name and the value indicated in the value expression. The operators currently defined are limited to = and !=, but <, >, <= and >= have been previously supported, and may be re-instated in a future version. Their semantics (discussed below) depends on the nature of the value expression.

3.1.3 Value expressions

Value expressions may be unquoted or quoted strings or numbers, or disjunctions of strings or numbers. The types of expressions allowed are listed in Table 3.3.

3.1.4 Operator semantics

Table 3.4 summarises the semantics of the operators by stating the result when the operator is applied between arguments of each type, including variables. In the table, the column \(A\) refers to the value of the feature named \(A\) for the text unit currently being compared. For example, in the expression \(\text{token}="\text{new}\"\), \(A\) is shorthand for the value of the token feature of the text unit currently being compared with this rule constituent, whilst “\text{new}\” is what the rule says it ought to be, for the rule to succeed.

\(^1\)Feature names are case-insensitive, so sem is equivalent to SEM.
Table 3.1: The basic features established by earlier analysis phases

- **token**: The string representation of the text span.
- **orth**: A code for the orthographic pattern of a token. Values will depend on the tokenizer used. With the CAFETIERE tokenizer, these codes are as listed in Table 3.2. **NB** The codes differ from those produced by the GATE tokenizer.
- **pos**: This feature accesses the part of speech tag assigned by the part of speech tagger. With the Brill tagger currently in use, the pos feature is singly-valued, but in a previous version, a tagger was used which assigned composite tags which were represented as multiple values, e.g. NN and PL for a plural common noun. **NB** The syntactic phrase type is indicated by the feature syn when it is set by rule. (The noun phrase, for example, is not one of the parts of speech, which are the labels that classify words.)
- **lem**: If a morphological analyser is included in the system configuration, the lem value will be the morphological root, otherwise it will be the token, normalized to lower case. Currently, there is no morphological analyzer. However, morphological features are encoded in some part of speech tags (e.g. VBD, VBG, VBN etc.).
- **stem**: If a stemmer is used as a substitute for a morphological analyser, this feature will be present for each text token, and will provide the stem as determined by the selected stemming algorithm. Currently, there is no stemmer in the configuration, but one is planned for a future version.
- **lookup**: This feature is selectively assigned to text elements by gazetteer or knowledge-base lookup. Tokens and phrases may be ambiguous (polysynonymous), e.g. “Washington” can be two sorts of place and a surname. The multiple entries in the gazetteer are translated to a multi-valued list associated with the text unit in memory during rule-based analysis. As with the pos and syn features, there is a different label, sem, for the comparable information when set by rule.
- **ante**: This feature was used in previous versions to identify a textually antecedent element that has been found to be co-referent with the element. It may be re-instated in the current version, but will probably get the name coref, since co-referent items can follow as well as precede the referring expression.
- **id**: This feature stores a unique identifier for the token in the text. A new sequence of identifiers is started for each text processed.
- **zone**: If there is a smart document import facility, it may be able to separate the text into front matter (such as document id, dateline and by-line) and body text. The system controller may be able to invoke the analyser separately with distinct rule sets for different zones. However, an alternative strategy is to have rules specify the zone to which they apply as just another feature constraint. Furthermore, rules can constrain all constituent text elements to belong to the same zone, to avoid accidental spurious matches. If there is not a text zoning component, all text elements default to the zone ‘body’.
Table 3.2: Codes for the orth feature and their meanings

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowercase</td>
<td>Every character is a lowercase letter.</td>
</tr>
<tr>
<td>capitalised</td>
<td>The first character is an uppercase letter; all other characters are lowercase letters.</td>
</tr>
<tr>
<td>caphyphenated</td>
<td>As above, but there is one or more hyphens among the characters.</td>
</tr>
<tr>
<td>lhyphenated</td>
<td>A lowercase hyphenated word.</td>
</tr>
<tr>
<td>uppercase</td>
<td>Every character is an uppercase letter.</td>
</tr>
<tr>
<td>multicap</td>
<td>There is more than one uppercase letter in the token.</td>
</tr>
<tr>
<td>upperdotted</td>
<td>The token contains only uppercase letters and dots.</td>
</tr>
<tr>
<td>initial</td>
<td>The token is a single uppercase letter.</td>
</tr>
<tr>
<td>initialdot</td>
<td>The token is a single uppercase letter followed by a dot.</td>
</tr>
<tr>
<td>arithmetic</td>
<td>The token is an arithmetic or algebraic operator.</td>
</tr>
<tr>
<td>doublequote</td>
<td>The token is a double quotation mark.</td>
</tr>
<tr>
<td>apostrophe</td>
<td>The token is the apostrophe/single quotation mark.</td>
</tr>
<tr>
<td>number</td>
<td>The token is a number.</td>
</tr>
<tr>
<td>bracket</td>
<td>The token is an opening or closing bracket, brace, parenthesis, etc.</td>
</tr>
<tr>
<td>punct</td>
<td>The token is a punctuation character.</td>
</tr>
<tr>
<td>other</td>
<td>The token is any sequence of characters that does not match any of the above descriptions.</td>
</tr>
</tbody>
</table>

3.2 Exploiting the rule notation

Examples of different usages of the formalism are given below. They are grouped according to the features of the formalism that they exploit. Note that in a rule file, lines that commence with the # symbol are treated as comments. The comment lines preceding a rule are read by the rule parser, and are available to view when the rules are browsed or edited in a tree view rule display.

3.2.1 A simple context free rule

Example (3.7) is about the simplest usage of the rule formalism. It would match an expression such as “40 mg” where a cardinal number (syn=CD) is followed by a word that the lookup resource tags as a measure.

(3.7) \[syn=np, sem=qty\] => \\[syn=CD], \[sem=measure\] / ;

Rule (3.7) is a simple context free rule, in that both constituents (and nothing else) are required for the rule to succeed. A rule is context-free if it has no text units to the left of \ and none to the right of /. When the rule is satisfied, a new phrase is created with the syn feature np and the sem feature qty, and the new phrase replaces its constituents in the token sequence.
Table 3.3: Types of values associated with features

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unquoted strings</strong></td>
<td>A hangover from a previous Lisp implementation, where they were converted to the symbol data type. This type of value expression is normally used where the domain is a small set of values, such as the parts of speech, or morphological features.</td>
</tr>
<tr>
<td><strong>Quoted strings</strong></td>
<td>Strings denote string literals, and may contain the wildcards “*” and “?” for ‘glob’-style pattern matching. Wildcards may be escaped by a preceding “\”.</td>
</tr>
<tr>
<td><strong>Numbers</strong></td>
<td>In the current implementation, numbers are simply treated as unquoted strings, but this may change in a subsequent version.</td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td>Variables are written as a special case of unquoted string, having an initial underscore. They are treated like Prolog variables, sharing with other occurrences of the same variable, and have scope and extent limited to the current rule and its invocation. A variable is instantiated if a value is associated with it. An uninstatiated variable is one that does not, at a given point in the rule’s invocation, have a value associated with it. Unlike variables in procedural languages, these variables may not be re-assigned with new values while the rule is being applied to the same text span. When a variable becomes instantiated, all instances of the same variable become implicitly associated with that instantiation. Illustrations of the use of variables are given below.</td>
</tr>
<tr>
<td><strong>Disjunctions</strong></td>
<td>To avoid the necessity for multiple rules that differ only in specific feature values, a single value expression may denote a disjunction of values. These are indicated by the use of the pipe symbol,</td>
</tr>
</tbody>
</table>
### Table 3.4: Operator semantics

<table>
<thead>
<tr>
<th>expression</th>
<th>$A$</th>
<th>$B$ or $B_i$</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>on the right-hand side of a rule</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A = B$</td>
<td>string</td>
<td>simple quoted string or unquoted string</td>
<td>true iff $A$ and $B$ are equal strings, ignoring case.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>string</td>
<td>quoted string with wildcard(s)</td>
<td>true iff $B$ matches $A$, with every ? in $B$ matching the character in the corresponding position in $A$, and every * in $B$ matching a sequence of zero or more characters from $A$ starting at the position in $A$ corresponding to that of the * (taking into account the matches of all previous * in $B$).</td>
</tr>
<tr>
<td>$A = B$</td>
<td>list of strings</td>
<td>simple quoted or unquoted string (possibly containing wildcards)</td>
<td>true if $B$ matches (ignoring case) any member of $A$ as in either of the foregoing cases.</td>
</tr>
<tr>
<td>$A = B_1</td>
<td>B_2 | \ldots</td>
<td>B_n$</td>
<td>string</td>
</tr>
<tr>
<td>$A = B_1</td>
<td>B_2 | \ldots</td>
<td>B_n$</td>
<td>list of strings</td>
</tr>
<tr>
<td>$A = B$</td>
<td>no value</td>
<td>uninstantiated variable</td>
<td>true.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>no value</td>
<td>instantiated variable</td>
<td>true.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>any type</td>
<td>uninstantiated variable</td>
<td>true, and as a side effect, the variable becomes instantiated to the value $A$.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>string</td>
<td>variable instantiated to a string</td>
<td>true iff $A$ matches the instantiation of $B$, ignoring case.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>list of strings</td>
<td>variable instantiated to a list of strings</td>
<td>true iff each member of $A$ matches the corresponding member of the instantiation of $B$, ignoring case.</td>
</tr>
<tr>
<td>$A \neq B$</td>
<td>simple string or number or variable</td>
<td>simple string or number or variable</td>
<td>true iff $A$ and $B$ fail to match, ignoring case, i.e. it is not the case that $A = B$. (There is no propositional negation operator.)</td>
</tr>
<tr>
<td>$A \neq B$</td>
<td>list of strings or number or variable</td>
<td>simple string or number or variable</td>
<td>true iff $B$ fails to match with any member of $A$, ignoring case.</td>
</tr>
<tr>
<td>on the left-hand side of a rule</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A = B$</td>
<td>no value</td>
<td>string</td>
<td>true, and by side effect, a feature named “$A$” in the newly created phrasal text unit is associated with the value $B$.</td>
</tr>
<tr>
<td>$A = B$</td>
<td>no value</td>
<td>instantiated variable</td>
<td>true, and by side effect, a feature named “$A$” in the newly created phrasal text unit is associated with the value of the instantiation of $B$.</td>
</tr>
</tbody>
</table>
3.2.2 Regular expression operators and ranges

A text element description on the right-hand side of a rule may be followed by ?, * or +, showing that it is optional, may be matched from zero to an infinite number of times, or may be matched from 1 to an infinite number of times, respectively. Alternatively, precise lower and upper limits to its iteration may be specified as a pair of numbers in braces, e.g. \{2,4\} for a text element description that must be matched at least twice in succession and may be matched up to four times in succession. The use of iteration ranges is preferred over the unlimited iteration operators * and +.

3.2.3 Grouping of constituents

Sequences of text unit descriptions may be grouped by being enclosed in round brackets. Regular expression operators or ranges may be placed both after the individual text units and after grouping parentheses. An example of such a sequence is a list, where word sequences are separated by commas, as in (3.8) or (3.9), which are intended to cover the same type of phrase.\(^3\)

(3.8) \[syn=NNPS,sem=NAMELIST\] =>
\[orth=capitalized]+, ([token=","], [orth=capitalized]+)+ / ;

(3.9) \[syn=NNPS,sem=NAMELIST\] =>
\[orth=capitalized]\{1,2\},
([token=","], [orth=capitalized]\{1,2\}\{1,9\} / ;

3.2.4 Pure grammar rules and semantic grammar rules

The formalism can be used to write a full or partial syntactic grammar of a language. The rules in such a grammar would typically concentrate on the use of the \textit{syn} feature on both left and right-hand side of the rule. Rule (3.10) has only syntactic conditions. On the other hand, for most information extraction applications, one is more likely to be interested in the semantic classification of analyzed fragments, as in rule (3.11). Both rules make use of disjunction of values, and both make use of the regular expression operator *, by which the preceding text unit is permitted to occur zero or more times.

(3.10) \# A syntactic rule to identify a noun group
\[syn=NG, rulid=NounGrp1\] =>
\[syn=NN|JJ|JJR\]*,
\[syn=NN|NNS\] /;

(3.11) \# A pre-modified food-drug-product is a product
\[syn=NG, SEM=PRODUCT, ZONE=BODY, RULID=Prod_1\] =>
\[syn=NN|JJ, zone=BODY\]*,
\[sem=FOOD-DRUG-PRODUCT, zone=BODY\] /;

Rule (3.10) is a simple syntactic rule to match any a sequence of zero or more singular common nouns (tagged NN),\(^4\) adjectives (JJ) or comparative adjectives (JJR), followed by a

\(^3\)Bear in mind that this example is given to illustrate the formalism. It will not be a good analysis if the purpose of the analysis is to extract the individual members of the list of names.

\(^4\)These tags are those of the Penn treebank, as used in our version of the Brill tagger, but other tagsets could be used.
final singular or plural (NNS) common noun. The resulting phrase is labelled syntactically (by
the feature syn) as a NG.\(^5\). The value of the feature rulid identifies the rule for debugging
purposes. This feature’s value will be among the features of the resulting annotation, and
can be browsed through the annotation viewing facilities, and searched for in the text editor
in which the rules are being edited.

(3.11) is a more typical rule used in an information extraction application. This one
labels a phrase as a PRODUCT if it contains pre-modifying nouns or adjectives and a text
span already labelled as a FOOD-DRUG-PRODUCT. The label FOOD-DRUG-PRODUCT
should have been associated with the text span in the gazetteer or knowledge-base lookup
phase. As explained in Table 3.1, we currently use the same label sem for items that have been
semantically classified by rule and by lookup. In the previous Lisp version of the software,
lookup items had a value for the source feature of db or mwdb, should the rule writer wish
to constrain a rule like (3.11) to apply only to items semantically labelled as FOOD-DRUG-
PRODUCT by lookup.

3.2.5 Variables and their uses

Rule (3.10) has a shortcoming, in that it can result in the loss of information about semantics.
If the last noun has a known semantic category, as encoded by the lookup feature, it should
be identified as the semantic category of the whole phrase. The trouble is, we don’t know
what that category is going to be. One way round this would be to write a rule like (3.12)
for each possible semantic category.

(3.12) # A syntactic rule to identify a noun group
      [syn=NG, sem=PRODUCT,rulid=NounGrpProd1] =>
      \ [syn=NN|JJ|JJR]*,
      [syn=NN|NNS,lookup=PRODUCT] /;

(3.13) # A syntactic rule to identify a noun group
      [syn=NG, sem=_sem,rulid=NounGrpVar] =>
      \ [syn=NN|JJ|JJR]*,
      [syn=NN|NNS,lookup=_sem] /;

In some applications, especially if lookup is connected to a general-purpose knowledge-base,
there could be a very large number of semantic categories, so this approach is tedious and
inefficient.

The solution is to use variables, which can range over all semantic categories. Within
a rule, each occurrence of the variable has the same instantiation. In Rule (3.13), we can
generalize over products, species, weapons or whatever, with the variable _sem. Whatever
instantiation _sem receives when the last constituent of Rule (3.13) is matched with a text
unit will be the value of the other occurrence on the left hand side of the rule. In this way
the instantiation will become the _sem feature value of the phrase recognized by the rule. This
allows the rule to capture the linguistic generalization that the semantics of a noun group as
a whole is essentially that of its head noun. In English and other languages that are said to
be “head-final”, the head of a noun phrase or group comes last.

\(^5\)This, being a label for a nonterminal phrase category, is not among those of the tagset. It is up to the
rule writing team to define a set of syntactic phrasal categories to use consistently.
In summary, Rule (3.13) shows one use of variables: to copy a feature value from a constituent to the phrase that includes it.

Another use of variables is to constrain the text elements within a phrase to share the same value for a feature. The classic application of this in full parsing is to enforce agreement (e.g. in number, between the subject and verb, or between determiner and head noun). This can be done by having the same constraint, e.g. `num=_N`, in the two constituents on the right-hand side of a rule.

### 3.2.6 Illustrations of the use of features

Features encode properties of text spans. They should be chosen for their relevance to the type of analysis being undertaken. As an example, suppose there is no morphological analyser in the configuration. Morphological features encoded by the part of speech tag could be made explicit by rules such as (3.14) and (3.15).

\[(3.14) \quad \text{[syn=NN, num=SG]} \Rightarrow \text{[syn=NN]} / ; \]
\[(3.15) \quad \text{[syn=NN, num=PL]} \Rightarrow \text{[syn=NNS]} / ; \]

Another illustration of the use of features is where one proper name is embedded in another, and we want to identify the embedded name as a property of the whole name. Rule (3.16) identifies the name of a university and records its location as a feature.

\[(3.16) \quad \text{[syn=NP, sem=ORG, sector=EDU, loc=_LOC]} \Rightarrow \]
\[\text{[token="University"],} \]
\[\text{[token="of"],} \]
\[\text{[sem=LOC, token=_LOC]} / ; \]

In this example, it is the string that names the location which becomes the value of the loc feature, through the shared _LOC variable.

This rule illustrates several other points of note:

- The symbolic value LOC should not be confused with the variable _LOC, nor with the feature named loc. (Feature names and value names are distinct namespaces that will not be confused by the rule compiler.)
- Rules can use a mixture of specific tokens and generic categories in their evidence for phrasehood.
- The use of the generic class org and the separate feature sector is not the only way to represent the conceptual class of a university. The ontology implied by this example is one where differentiating features are preferred over subtypes. The policy on the use of features should be compatible with whatever external gazetteer or knowledge-base is accessed prior to rule-based analysis.

### Standardization of features

Although the basic features listed in Table 3.1 are built into the configuration of the software, there is no restriction on the names that may be used for features on the left-hand side of rules. Groups of people working together on rule development will need to impose their own
discipline on the use and naming of features. This is not just a matter of making the end results of the analysis intelligible, since the output of a rule may be used in a condition of another rule. Introducing a feature needs to be done with care, and documented.

### 3.2.7 Use of wildcards in string matching

Partial matching of strings can be a useful heuristic if the rule-writer is prepared to sacrifice some precision for recall. For example, many surnames begin with the prefix “Mac” or “Mc”. Rule (3.17) shows how this can be written:

\[
\text{(3.17)} \quad \left[ \text{syn=NNP, sem=SURNAME} \right] \Rightarrow \left[ \text{token="Mac??*"|"Mc??*", orth=capitalized|multicap} \right] / ;
\]

A point to note in relation to this example is that the matcher will ignore case, so the pattern would get false positives like “machine”, unless further constrained. The use of the orthography feature is important here. The orthography pattern `capitalized` would allow examples like “Macaulay” but not “machine” but would also miss “MacEwan”, hence the alternative `multicap`.

### 3.3 The use of context

Names of various kinds often have prefixes or suffixes that are not normally considered part of the name. Personal titles, such as “Miss”, “Dr”, “Gen.”, are a case in point. These provide a very good clue to the status of following capitalized words that may not be in the gazetteer. An example rule relying on the previous context in this way is (3.18), which assumes prefix titles to be in the gazetteer or knowledge-base.

\[
\text{(3.18)} \quad \left[ \text{syn=NNP, sem=PERSON} \right] \Rightarrow \left[ \text{sem=title}\{1,2\} \right.
\left. \text{orth=capitalized},
\text{ortho=upperinitial}?,
\text{ortho=capitalized} \right] / ;
\]

When writing rules with contexts, the contextual constraints should not be entirely optional, because if they can be omitted, they need not be there, and their presence can mislead the rule-writer into thinking the rule is more precisely applicable than it in fact is. In (3.18), the title is partly optional, in that there can be either one or two title tokens for the rule to apply.

### 3.4 Co-reference between phrases

Rule schema (3.2) – repeated for convenience as (3.19) – provides a way to express the co-reference between spans of text \( C \) (which, if the rule succeeds, is labelled and described as \( A \)), and \( E \) (the antecedent).

\[
\text{(3.19)} \quad A => B \backslash C/D >> E;
\]
It is the rule-writer’s responsibility to ensure there is some basis for asserting co-reference. This rule schema was originally intended to classify text spans as names on the basis that the current span is identical to or a sub-string of the antecedent.

To express the constraint that the span \( C \) should be the same string as the antecedent \( E \), both \( C \) and \( E \) should include the condition \( \text{token} = \text{Var} \), where \( \text{Var} \) is a shared variable. Such a rule should be used selectively, normally only when \( E \) also satisfies conditions such as \( \text{syn} = \text{NNP} \), \( \text{sem} = \text{person} \), since repeated use of categories other than proper names, e.g. common noun (NN), does not denote co-reference.

Where \( C \) is a shorter form of its antecedent \( E \), co-reference rules will only work if the parts of the antecedent have been selected as feature values. Rule (3.20) anticipates the needs of subsequent co-reference rules by assigning the first name to the feature \( \text{firstname} \) and the surname to \( \text{surname} \). Rule (3.21) can identify a subsequent instance of the surname on its own as coreferent, and hence of the same category \( \text{PERSON} \).

\[
(3.20) \quad \left[ \text{syn} = \text{NNP}, \text{sem} = \text{PERSON}, \text{firstname} = _F, \text{surname} = _S \right] \Rightarrow \\
\quad \left[ \text{orth} = \text{capitalized}, \text{sem} = \text{person\_male}, \text{token} = _F \right], \\
\quad \left[ \text{orth} = \text{capitalized}, \text{token} = _S \right] / ;
\]

\[
(3.21) \quad \left[ \text{syn} = \text{NNP}, \text{sem} = \text{PERSON}, \text{firstname} = _F, \text{surname} = _S \right] \Rightarrow \\
\quad \left[ \text{orth} = \text{capitalized}, \text{token} = _S \right] / \\
\quad \quad >> \\
\quad \left[ \text{sem} = \text{PERSON}, \text{firstname} = _F, \text{surname} = _S \right];
\]

In earlier versions of the analyser, a rule like (3.21) found a single antecedent text unit. In case of multiple candidate antecedents, the most recent was selected. Co-reference was not established merely in order to confirm the \( \text{sem} \) value of the consequent phrase, but to identify the two phrases as denoting the same entity. This was done by the system creating a feature \( \text{ante} \) in the newly added phrase whose value was the unique identifier of the antecedent text unit.

Since the coreference relation is a transitive, indeed an equivalence relation, it was possible, post rule application, to identify the set of text units denoting an individual entity. The present version of the system finds all edges that satisfy the constraints expressed in \( E \), and during rule application, creates an entity as its result, whose features are the union of the attribute-values of all instances of \( E \) together with \( C \), and whose members are all their phrase identifiers.

Another way in which the current version of the system differs from its predecessors is that co-reference can be established between a text span and others that textually follow it rather than precede it. This can happen within the sentence, as in (3.22), but in news texts it often happens that names used in titles or headings are briefer than those found in the first sentences of the main body of the text.

\[
(3.22) \quad \text{Near him}, \text{John} \text{ saw a snake.}
\]

### 3.4.1 Potential of coreference

With the mechanism provided, there is scope for finding more than co-referent person names. In the Unilever scientific article corpora, it should be possible to track literature citations and to associate sentences containing them with specific bibliographic entries, for example.
It is also planned to experiment with the formalism’s potential for keeping track of pronoun co-reference, e.g. by using the heuristic approach described in [11]. To implement this fully, the rule formalism would ideally be extended to use scoring for different rules as a means of selecting preferred antecedents rather than only the most recent candidate referent.

3.4.2 Coreference and apposition

Apart from coreference between non-adjacent text units, natural languages also express implicit coreference in appositive structures. This type of co-reference is local in its scope, and does not require the mechanisms described earlier in this section.

Two noun phrases are said to be in *apposition* if they are textually adjacent and co-referent. Typically one of the two is a proper noun, and the other a definite description, or one a proper noun and the other an abbreviation or an acronym. In apposition, the second noun phrase is set off from the surrounding text, conventionally by parentheses in the case of abbreviations, otherwise by enclosure in a pair of commas. Examples include (3.23a), (3.23b) and (3.23c), the last of which can be recognized with rules (3.24a) and/or (3.24b).

(3.23)  
\[\text{a. The Foreign Secretary, Jack Straw,} \]
\[\text{b. Jack Straw, the Foreign Secretary,} \]
\[\text{c. The Food and Drug Administration (FDA).} \]

(3.24)  
\[\text{a. [syn=NNP, sem=ORG_GOV, abbrev=_A] =>} \]
\[\text{[syn=DT],} \]
\[\text{[orth=capitalized]{1,3},} \]
\[\text{[syn=CC]?} , \]
\[\text{[orth=capitalized]{1,2},} \]
\[\text{[orth=capitalized,sem=ORG_HEAD] /} \]
\[\text{[token="\),} \]
\[\text{[orth=uppercase,token=_A],} \]
\[\text{[token=""]}\] ;

\[\text{b. [syn=NNP, sem=_SEM, ante=_ANTE] =>} \]
\[\text{[sem!=NIL, sem=_SEM, id=_ANTE]} \]
\[\text{[token="\),} \]
\[\text{[orth=uppercase],} \]
\[\text{[token=""]}\] / ;

However, it can happen that an item in parentheses is something other than an abbreviation or acronym for the phrase it follows. It is planned therefore to permit a rule to require additional evidence of an abbreviation. This will be possible if a text unit has a feature *initial*, assigned on text read-in, whose value is the first letter of the token.

In support of this feature, it is also planned in the version of the system that accesses the knowledge-base to map both abbreviations and full forms of equivalent known terms to the same object Id.

---

\[\text{\textsuperscript{6}This is given a different analysis when ‘Foreign Secretary’ is not marked as definite, as in the US English usage where occupational titles are more often used as honorifics than in British English. If ‘Foreign Secretary’ is an honorific rather than an appositive, there is no comma separating it from the name.}\]
3.5 Rules to create entity instances

In the Parmenides common annotation scheme, a distinction is made between textual and conceptual annotations. Textual annotations are placed in the text, but conceptual annotations are not. The former provide textual evidence on which to base the creation of the latter.

In the Cafetiere system, there has to be a way to distinguish a conceptual annotation. To do so, the left-hand side of a rule must include the assignment of the feature label to a text string which is the token value of a phrase already created as a lexical annotation.\footnote{When the knowledge-base-linked version of the system is in place, the entity.id may be a more appropriate identifier for the entity annotation.} After all rules have been applied, conceptual annotations are created with all instances sharing the same label assimilated to a single entity. The entity created has the union of the features of its member lexical annotations.

For this to work with phrases or more than one constituent, the rule that sets the label cannot be the same one that recognizes the phrase as a lexical annotation.
Chapter 4

Processing and Rule Order

When the processor is invoked on a text, the rules are applied strictly in sequence, and the words and phrases identified by rule replace their constituents, which are no longer ‘visible’ to subsequent rules.

An earlier version behaved as a standard left-corner chart parser, where rule ordering is not significant. The different versions applied to the same rule set and text can therefore achieve different results. The chart version has been discontinued because it is slower, and also, being deterministic, requires heuristics to select between competing overlapping analyses. Now, the only way of influencing which of two competing rules’ analyses should succeed on a given edge sequence is rule ordering.

To illustrate the effect of rule ordering, let us consider a concrete example of a short text (4.1). It will be seen that the punctuation symbols are tokens, but “Ph.D.”, which clearly includes punctuation symbols, is shown as a single token. This is assumed to have been identified as a token either by the tokenizer, or by the lookup module.

(4.1)

\begin{tabular}{cccccccc}
\textit{token}: & Charles & R. & Wescott, & Ph.D., & Senior & Scientist & at & Dyax Corp \\
\textit{syn}: & NNP & NNP & NNP, & NNP, & JJ & NN & TO & NNP NNP \\
\textit{sem}: & qual & job & co_head
\end{tabular}

Suppose we have the rules (4.2) and (4.3):

(4.2) \[
[syn=nn] => \\
\quad [syn=JJ\{NN\}\{1,3\}, \ [syn=NN] \ / \\
\]

(4.3) \[
[syn=nn,sem=job] => \\
\quad [syn=JJ\{NN\}\{1,3\}, \ [sem=job] \ / \\
\]

Rule (4.2) will match the text items “Senior” and “Scientist” to rewrite the token sequence as (4.4).

(4.4)

\begin{tabular}{cccccccc}
\textit{token}: & Charles & R. & Wescott, & Ph.D., & Senior & Scientist & at & Dyax Corp \\
\textit{syn}: & NNP & NNP & NNP, & NNP, & NN & TO & NNP NNP \\
\textit{sem}: & qual & job & co_head
\end{tabular}

Since “Senior Scientist” is now a single text unit and furthermore, there is no text unit labelled \textit{sem=job}, rule (4.3) cannot now succeed. However, if (4.3) precedes (4.2), the former will succeed, blocking the latter.

As a general guide to the construction of rule sequences, rules with more specific constraints should be placed before those with less specific constraints.
To mitigate the effects of rules blocking later rules when they should not (and at the same
time ensuring better precision), contextual constraints should be provided wherever possible.
(Recall that there is no point in having contextual constraints that are optional.)

Another policy that tends to coincide with the precedence of more specific rules is that
rules that are more likely to be reliable should precede those that are more heuristic. Rules
tend to be more reliable where they depend on prior lookup of text units, or when contextual
constraints have been set.

Some policies for rule ordering can be in conflict. For example, rules that establish the
class of a text span through co-reference with other spans are quite reliable, but since they
depend on the prior success of other rules, their placement should be relatively late in the
sequence. It can even be sensible to have the same rule twice in the rule sequence, since
intervening rules could create its prerequisite constituents.

4.1 Recursive rules

Different parsing algorithms are sensitive to recursive rules in different ways. The bottom-up
procedure used here can be non-terminating when applied to a rule of the form (3.1), where
the right-hand side $C$ subsumes the left-hand side $A$. A feature-value description is said to
subsume another if the latter has all the features specified by the former. For this to be a
potential problem, the right-hand side would have only a single constituent, so that as soon
as a rule has applied to a text span, it becomes immediately applicable to itself. This is the
case with rule (3.14), but not with (3.15), which has different values for the syn feature on
its left- and right-hand sides.

The default rule invocation mechanism checks for and prevents runaway recursion, but
since this is at a cost, it may be necessary, for efficiency, to provide the alternative of turning
off such checks, in which case the rule writer would need to add conditions that could not
be satisfied by the resultant phrase, e.g. num=null in (3.14). That there are separate feature
names for the syntactic and semantic properties of text units in different levels of analysis
will assist in avoiding runaway recursion. The tagger’s parts of speech are now labelled pos,
whereas the syntactic phrase labels assigned by rules (i.e. nonterminals) should be labelled
syn.\(^1\)

4.2 Repeatedly applicable rules

In a cascade of finite state rules, once a rule has been applied to a text, it cannot be be
reapplied if the result of the rule makes it applicable again. However, in the cafetiere
system, rules can be reapplied immediately. An example of a well-motivated rule like this
would be one to identify members of a list as all belonging to the same semantic class. Rule
(4.5) recognizes capitalized word, initial sequences as authors, when they are preceded by
authors.

\begin{verbatim}
(4.5)  [syn=nnp,sem=author] =>
   [sem=author], [token="","and"]
   \ [orth=capitalized], [orth=initial] / ;
\end{verbatim}

\(^1\)The label can be anything at all, as long as it is not pos.
4.3 Phrase, entity, relation and event rule order

Rules should be assembled in the order: phrase and entity, relation and event. Entity descriptions are made via the labelling of spans as noun- or proper noun phrases. To create an entity representation, a co-reference rule or rules establishes that a set of spans name the same entity.

Relations are relations between entities, and hence rules to identify them come after entity rules.

Events can only be established when all their constituent parts (or arguments) are analysed.
Chapter 5

Running the Analyser and Annotation Editor

The analyser is available for running both within the GATE environment and with its own user interface that supports the viewing and editing of annotations. The overall design and presentation of the annotation editor is very similar to GATE, so making the transition from one to the other should be very simple. The annotation editor is, however, a simpler interface compared with GATE since it does not attempt to support the engineer of a text analysis application by manipulating component tools. It has the core functionality that is needed by two classes of user:

- An end user, who wants to analyse some texts, either manually or by running the automatic tools, correcting and saving the results.

- A development user, who wants to perfect the ability to automatically analyse texts by evaluating the output, making changes to rules or knowledge base and evaluating again.

Figure 5.1 shows the layout of the user interface as it appears following the automatic analysis of a text. It has four component areas, one to the top left, which shows the text of the document, with highlights displaying selected element types. To the top right is the annotation tree area, which shows an expandable summary of the annotations created so far on the document. Below those two areas are two horizontal divisions. The upper of the two is a form in which a single annotation’s features may be edited, and the lower is a table that shows all of the features of the type currently selected in the annotation tree pane.

So far, quite like GATE, except that the feature editor is in the same window as the viewable elements, and that the features that can be edited are constrained. Unlike GATE, there is currently no notion of a corpus as such, and the editing or analysis can be initiated on any text in CAS format (as created by the document collector) through a standard file open dialog.
Figure 5.1: Annotation Editor interface
With a document loaded, the analysis can be initiated using the actions menu (Figure 5.2). In the choose actions check-box group, checking a box selects all the prerequisite analyses above it. When the processing is done, there will be new annotations (additional to the structural ones created by the document collector) summarised in the tree view.

Any of those can be viewed (as in GATE when its “Annotation Sets” and “Annotations” tabs have been selected) by selecting annotation groups or instances in the tree or tabular views. Figure 5.3 shows how selecting a class of lexical annotation causes its instances to be highlighted in the text and the features of those instances displayed in the tabular view below. Annotations of all classes, structural, lexical and conceptual (including entities, relations and events) can be viewed in essentially the same way.

5.1 Adding annotations manually

Annotations can be made from text spans selected by the mouse. Right-clicking produces a menu of the classes of lexical annotation available for selection, as shown in figure 5.4. The classes shown in the menu are those that are relevant to the application. There is no way through the interface to add a lexical annotation with any other class name. The available classes are set in the Common Annotation Scheme DTD (although in the final version, this will be done in the ontology of classes or of events).

5.1.1 Entity annotations

In the Annotation Editor, any annotation that can be created by automatic analysis can also be created manually. Starting from lexical annotations on which it is to be grounded, an entity annotations can be created by menu selection. The normal way to do this is to select an existing lexical annotation, and use the right-click menu shown in figure 5.5 either to create a new entity annotation with the lexical annotation as its first member, or add the lexical annotation to an existing entity.
Figure 5.3: Viewing the lexical annotations of one class

Figure 5.4: Adding a lexical annotation
Figure 5.5: Adding a lexical annotation to a new or existing entity annotation
Since all conceptual annotations are built on lexical annotations, the interface does not permit the creation of a conceptual annotation except via a lexical annotation.

As with lexical annotations, when a single annotation is selected, its features are loaded into the Feature Editor Area that was shown in figure 5.1, where their textual values can be changed (or entered, if the annotation is being newly created).
Chapter 6

Knowledge-base in place of gazetteer

This chapter describes planned rather than current capabilities of the Cafetiere system. It was noted earlier that it is a defect of the current version of the analyser that it does not import information about known individual entities when accessing the gazetteer. It is thus incapable in principle of building representations in a knowledge representation language like NKRL [21]. The main prerequisite of such a knowledge representation is that known individual objects are represented by the same unique identifiers that they have in the knowledge base. These identifiers can only come from reading the information from the knowledge repository.

6.1 Role of the knowledge-base in analysis

With a knowledge base in place of a gazetteer, the lookup stage of analysis can do more than before:

1. As with the gazetteer, supply semantic classes (concepts) corresponding to words and phrases.

2. Supply object identifiers for known instances, including where aliases and abbreviations name the same instance.

3. Supply properties of known instances, for example the country of which a city is the capital.

4. Obtain properties from the class definition that must be instantiated for previously unknown instances.

5. For verbs of interest to the application: supply not only the concept, but also the constraints it imposes on the types of its arguments.

In sum, the knowledge-base lookup module’s job is to go from natural language to concepts (both entities and event template instances).
To illustrate how an event is analysed with the benefit of the knowledge-base, consider (6.1) in which “Washington” is a metonym\(^1\) for the US government. The NKRL\(^2\) representation is (6.2). In this semantic representation, which has the form of a template, each numbered unit is an atomic predication, having a predicate name, and named slots. There are seven distinct slot names in the NKRL formalism. The slot values are the identifiers of conceptual entities rather than natural language strings. \textsc{Washington\_adm} is the identifier of the conceptual representation of the US government. The concept identifiers, whether denoting concrete instances like \textsc{Washington\_adm} or generic concepts like \textsc{Treaty}, exist within the background knowledge in the hierarchy of classes, which is referred to as \textsc{h\_class}. The predicates, together with their slot names and type constraints on slot values, exist in another hierarchy known as \textsc{h\_temp}.

(6.1) Washington announces it has signed the ABM treaty.

(6.2) \begin{itemize}
  \item \textsc{Move} \begin{tabular}{ll}
    \textsc{Subj} & \textsc{Washington\_adm} \\
    \textsc{Obj} & \#c2 \\
    \textsc{Date} &
  \end{tabular}
  \item \textsc{Produce} \begin{tabular}{ll}
    \textsc{Subj} & \textsc{Washington\_adm} \\
    \textsc{Obj} & \textsc{Signature1} \\
    \textsc{Topic} & \textsc{ABM Treaty}
  \end{tabular}
\end{itemize}

6.1.1 Step 1: Accessing properties of entities

When the lookup process accesses the knowledge-base with the proper name of a known entity, it should minimally expect to receive back two pieces of information: the class the entity belongs to, and its identifier.\(^3\) Where a text string names an object simply, the right-hand side of the rule will typically include the conditions \texttt{lookup= class, id=eid} to access these features. The class is most often used as a constraint on the rule, and the identifier is intended to be shared with a variable on the left-hand side of the rule.

The metonymy case is one where the access to the knowledge base is less straightforward. Without a knowledge-base, a rule to capture the metonymy would be rather under-constrained, as in (6.3), where any location is treated as representing a government if it is the subject of a finite verb.

(6.3) \[\text{[sem=GOV, syn=NP] =\> [syn=SENT] \} \text{[sem=LOCATION] / [syn=VBD|VBZ]}\]

\(^1\) Interpretation problems like this one are discussed in [8], which proposes to resolve such extensions of meaning by an abductive reasoning procedure. We do not use reasoning in the \textsc{Cafetiere} system, and so must conventionalize such inferences in rules.

\(^2\) Narrative Knowledge Representation Language. [21]

\(^3\) There are different systems for coining unique identifiers, but the NKRL convention for individual instances is to use the string representation augmented with at least one underscore. Where the name is a multi-word string, spaces are changed to underscores; where it is a single word string, an underscore is appended as a suffix. Numerical suffixes are not used unless there is a need to identify separately each of a set of homonyms.
With the knowledge-base, we should be able to know that “Washington” names a capital city, and that it it the capital of the USA, and to access the entity id’s of the relevant entities. This could be encoded without any enhancement to the notation, as in (6.4), provided the lookup mechanism is enhanced to read properties, like the country of a capital city, in addition to the class and identifier. The value of properties that represent relations with other entities, such as the country property for a city, would be the entity id of the relevant entity, in this case, that of of the USA. Other properties might have strings or numbers as their values.

Depending on the fastidiousness of the ontology, this approach might not be adequate, since it equates the country and its government. If the government of a country is a property of the country, then we need to be able to express the constraint that that country’s government is Gov, so that we can say in the left hand side of a revised rule eid = Gov. This amounts to the composition of the functions government:country→government and country:city→country.

To obtain the entity id of an entity that is the value of a property not of the current text unit, but of an entity represented by a variable appears to necessitate an enhancement to the rule notation. Two alternative ways in which the notation could be extended to allow such access to the properties of variable instantiations would be (6.5a) and (6.5b)

(6.5) a. [sem=capital_city,country=_country,_country.government=_gov]
    b. [sem=capital_city,country.government=_gov]

In both cases, we assume the property capital accesses the entity identifier of the country of which the object denoted by the current text unit is the capital. In (6.5a), we make explicit that the government property is only accessible once the country object has been accessed. (6.5a) relies on the analogy with function composition, but writes the functions in the opposite order to the mathematical notation for function composition which would be government◦country. In choosing between (6.5a) and (6.5b), it should be noted that the former would allow both variables to be accessed for use on rule l.h.s., whereas the latter reduces the prolixity of the notation. If the latter were adopted, it would still be possible to access the country variable by writing a separate feature=value equation.

Another enhancement seems desirable, and that is to be able to express class constraints at different levels of abstraction. If we represent the class of Washington, DC not as the atomic symbol CAPITAL_CITY, but rather as the path from the top of the knowledge-base of entities, e.g. LOCATION:CITY:CAPITAL_CITY, it would be possible for other rules that only cared whether the phrase designates a city, or even a location in general to succeed if the comparison is with a sub-path. We propose to use the operator >= for a comparison between the knowledge-base path (also known as the breadcrumb – Ellman, pers. comm.) of a text fragment and the one specified in the rule. For example, sem>=A:B:C would be satisfied if the text unit’s sem has A:B:C as a left sub-sequence. In practice we could get this effect by treating the knowledge-base paths as strings, in which case, the = operator and a final * wildcard could express the constraint in a way that can be implemented by the current machinery unaltered.
6.1.2 Step 2: Obtaining event constraints

Recall that (6.2) represented the event described in (6.1) with the predicates MOVE for “announced” and PRODUCE for “signed”. Each such predicate (actually their sub-classes) has constraints expressed in the h-temp on the type of argument each slot can take. When a verb (or event-denoting noun) is looked up, we need to ensure that those constraints can be applied to the subjects and objects, etc.

As a first attempt, a rule to build a CAS equivalent of the first part of (6.2) could look like (6.6).4

(6.6) \[
\text{[sem=EVENT:MOVE, subj=_S, eid=_e, obj=_O]} \Rightarrow \\
\text{[syn=NP, sem=SOCIAL_BODY:GOV, eid=_S]} \\
\text{[sem=EVENT:MOVE:SPEAK:ANNOUNCE, eid=_e]} / \\
\text{[sem<EVENT, eid=_O]} ;
\]

However, (6.6) suffers from the drawback that similar rules would need to be written for each event type of interest, although essentially duplicate work has gone into knowledge-base construction, describing the conceptual structure of each event class. If we can read from the knowledge-base the constraints on the subject and object (etc.) of an event, we might be able to write a single rule that will succeed when applied to any subject verb object combination. The problem with the way that constraints are set out in (6.6) is that they are expressed in the rule constituents representing the subject and object, whereas in the knowledge-base, they would be expressed in the verb’s conceptual structure. We need to be able to express the constraints within the verb’s constituent description.

A second attempt at such a rule could be (6.7), in which corresponding to each semantic role there is a feature instantiated at lookup time with the KB path (breadcrumb) to the class of the role filler.

(6.7) \[
\text{[sem=_esem, subj=_s, eid=_e, obj=_o]} \Rightarrow \\
\text{[syn=np, sem>=_ss, eid=_s]} \\
\text{[sem=_esem, eid=_e, subjsem=_ss, objsem=_os]} / \\
\text{[sem>=_os, eid=_o]} ;
\]

However, the notation could be better aligned with the knowledge base if the constraint on the arguments were expressed using the names of the slot and of the facet. Since (6.5a) uses the dot as a path separator, we need a different symbol, e.g. the tilde, to separate a slot name from a facet name—see (6.8).

(6.8) \[
\text{[sem=_es, subj=_s, eid=_e, obj=_o]} \Rightarrow \\
\text{[syn=np, sem=_ss, eid=_s]} \\
\text{[sem=_es, eid=_e, subj~sem=_ss, obj~sem=_os]} / \\
\text{[sem>=_os, eid=_o]} ;
\]

---

4In all of these examples, note that the context delimiters \ and / surround only the verb and not the whole phrase. That is because the intended result is a conceptual annotation and not a syntactic one, so there is no need to construct a text span that covers the whole sentence. The analysis is in the feature instantiations of the subj and obj, not in a syntactic bracketing. This is better from the point of view of the rule developer or end user browsing the results, since the annotations on the subject and object will remain visible.

5meaning one rule per syntactic arrangement of those constituents.
6.2 Knowledge-base lookup

To look up a word or phrase in an ontological resource augmented with a word→concept index,\(^6\) the algorithm is the same as looking up a gazetteer. There is the same need to look up multi-word phrases as well as single tokens. Similarly, the same natural language string can designate different entities. As with the present gazetteer lookup mechanism, most of the relevant data can be loaded from the knowledge base by the system before rule-application, although since more data about each string is loaded, the lookup mechanism could become more time-consuming.

However, there is one significant difference when we lookup multiple properties compared with looking up the single property of the category label from a gazetteer, as outlined in section 2.4.1.\(^7\) The representation of a text span will not be, as currently, as a single list of feature-value pairs where the values can be multiple, but rather as a list of lists of feature-value pairs. Each of the top-level list’s members will be an alternative complete characterization of the entity denoted by the text unit.

6.2.1 Need for analysis-time access to the knowledge base

The Parmenides and GATE architectures encourage NLP modules to be independent of each other, and a prior lookup process that looks up information about every text token means that the rule-based analyser need have no direct access to the gazetteer or knowledge base during rule-application.

However, where a semantic constraint involves a knowledge base class that is not instantiated by a text span directly, the need to load that class’s definition cannot be predicted by reading the text alone, which is what the current lookup module does. Either the knowledge base is consulted when a rule with a constraint like those in (6.5a) or (6.5b) is activated, or else the rules are scanned at lookup time, and the knowledge base is consulted for classes and their properties relevant to rules as well as to text items. These could be stored as Parmenides conceptual annotations, which differ from those created by analysis of the text by not having any member textual annotations. This probably implies extending the structure of conceptual annotations to cater for facets as well as feature values.

6.2.2 The knowledge-base lookup mechanism

To support the application of NLP rules, we need each concept label returned to be the ‘breadcrumb’ (the path from the top of the relevant hierarchy to that concept). This applies to the concept itself, and to the concepts that occur as type constraints in templates.

Lookup will work in two phases:

1. Retrieve all the NL phrases (only the text) that start with a given word. (The NLP lookup module will attempt to match these with the successive tokens of the text, filtering out all those that do not match, leaving exactly one string).

2. Retrieve all the NKRL representations as Java objects that are possible conceptual interpretations of a given string. The NLP lookup module then needs to be able to access

---

\(^6\)Typical knowledge base systems such as Protégé and NKRL do not have mappings from words to concepts, but conceptual dictionaries, such as WordNet and Wordmap, do.

\(^7\)This is not because of the information source being an knowledge-base, since it would apply if we were to look up both object id and category at the same time from a flat file or database table.
the conceptual predicate name, and all the slot names and argument type constraints per slot (or, in the case of instances under the H-CLASS, the concrete slot values).
Bibliography


