LEXICO-SEMANTIC PATTERN MATCHING AS A COMPANION TO PARSING IN TEXT UNDERSTANDING

Paul S. Jacobs, George R. Krupka and Lisa F. Rau

Artificial Intelligence Laboratory
GE Research and Development
Schenectady, NY 12301

ABSTRACT

Ordinarily, one thinks of the problem of natural language understanding as one of making a single, left-to-right pass through an input, producing a progressively refined and detailed interpretation. In text interpretation, however, the constraints of strict left-to-right processing are an encumbrance. Multi-pass methods, especially by interpreting words using corpus data and associating units of text with possible interpretations, can be more accurate and faster than single-pass methods of data extraction. Quality improves because corpus-based data and global context help to control false interpretations; speed improves because processing focuses on relevant sections.

The most useful forms of pre-processing for text interpretation use fairly superficial analysis that complements the style of ordinary parsing but uses much of the same knowledge base. Lexico-semantic pattern matching, with rules that combine lexical analysis with ordering and semantic categories, is a good method for this form of analysis. This type of pre-processing is efficient, takes advantage of corpus data, prevents many garden paths and fruitless parses, and helps the parser cope with the complexity and flexibility of real text.

INTRODUCTION

The interpretation of large volumes of text poses many control problems, including limiting the complexity of analysis and ensuring the production of valid interpretations without considering too many possibilities. These problems are especially severe in processing news stories, where long sentences, information-rich news-style constructions, and the complex structure of events make normal syntax-first analysis especially impractical.

Normal left-to-right syntactic parsing, in virtually all its forms, is a disaster for interpreting broad classes of extended texts. Multiple-path methods are haunted by attachment problems that can lead to a combinatoric explosion of paths, while simple deterministic methods bring on parser failures and problems in combining preferences. In previous work aimed at word sense coding of news stories [1], we have found that even heavy pruning of a multiple-path chart parsing strategy often leaves hundreds of parses to consider for a single sentence. Even worse, minor irregularities in linguistic structure or word usage bring on parser failures and inadequate interpretations.

Better parsing strategies, including control using statistical data, flexible partial parsing, and recovery, can certainly help with some of these problems, but some of the easiest improvements in the control of parsing come from the creative use of pre-processing. Our system incorporates a lexico-semantic pattern matcher, which uses much of the same knowledge base as the parser and semantic interpreter but performs a global, superficial analysis of text prior to parsing. The design and implementation of the pattern matcher is simple; instead of concentrating on its details, this paper focuses on the functionality of pre-processing and its impact on parser control.

Three aspects of pre-processing have particular promise for the quality and efficiency of later processing—*tagging*, *template activation* (including topic analysis), and *segmentation* (or bracketing). Tagging uses lexical data to constrain the part of speech and word senses of important words, template activation determines a set of possible templates, or frames, and segmentation associates portions of text with templates or template fillers. These techniques help the language analyzer to cope with the complexity of real text, both by reducing the combinatorics of parsing and by constraining word senses and attachment decisions. The following is a sample text taken from the development corpus of the MUC-3 message understanding evaluation [1], with the results of pre-processing after segmentation:

Original text:

SIX PEOPLE WERE KILLED AND FIVE WOUNDED TODAY IN A BOMB ATTACK THAT DESTROYED A PEASANT HOME IN THE TOWN OF QUINCHIA, ABOUT 300 KM WEST OF BOGOTA, IN THE COFFEE-GROWING DEPARTMENT OF RISARALDA, QUINCHIA MAYOR SAUL BOTERO HAS REPORTED. (41 words)

Segmented text:

[SIX PEOPLE] [A: WERE KILLED] AND FIVE [A: WOUNDED] [TIME: TODAY] IN [A: A BOMB ATTACK] THAT [A: DESTROYED] [A PEASANT HOME] [LOCATION: IN THE TOWN OF QUINCHIA] [DISTANCE: *COMMA* ABOUT 300 KM WEST OF BOGOTA] [LOCATION: *COMMA* IN THE

[1] MUC-3, the third government-sponsored message understanding evaluation, is in progress. Later in this paper, we will discuss the task and performance on the task.

COFFEE *HYPHEN* GROWING DEPARTMENT OF RISARALDA] [SOURCE: *COMMA* QUINCHIA MAYOR SAUL BOTERO HAS REPORTED] *PERIOD*

The label *A* in some segments indicates that those segments are template activators for a single event (single events are generally the default for multiple references within a sentence, unless there is a specific contextual cue such as a shift of time or location). The other labels are names of possible roles in templates. As is typical in news stories, roles can be shared (like time or location) or can apply to a single sub-event (like the number killed and wounded).

By grouping and labeling portions of text early, the program greatly reduces the amount of real parsing that must be done, eliminates many failed parses, and provides template-filling information that helps with later processing. For example, the phrase IN THE TOWN OF QUINCHIA is at least five ways ambiguous—it could modify A PEASANT HOME, DESTROYED, A BOMB ATTACK, WOUNDED, or WERE KILLED AND FIVE [WERE] WOUNDED. However, all five of these possibilities have the same effect on the final templates produced, so the program can defer any decisions about how to parse these phrases until after it has determined that the killing, wounding, attacking, and destruction are all part of the same event. Since these choices combine with the ambiguity of other phrases, the parsing process would otherwise be needlessly combinatoric. In fact, parsing contributes nothing after A PEASANT HOME, so this sentence can be processed as a 16-word example with some extra modifiers.

In addition to reducing the combinatorics of modifier attachment, pre-processing helps in resolving false ambiguities that are a matter of style in this sort of text. In this example, the ellipsis in FIVE [WERE] WOUNDED would be difficult, except that WOUNDED, like many transitive verbs, is never used as an active verb without a direct object. The ellipsis is thus detected prior to parsing, to be resolved during parsing rather than as part of recovering or detecting a syntactic gap. The early bracketing of the text allows the parser to resolve these complexities and ambiguities without much extra baggage, and without having to wait for a complete verb phrase.

Pre-processing not only speeds up parsing by avoiding combinatorics; it also improves the accuracy of interpretation, both by avoiding failures and by recognizing phrases and constructions that have specialized meaning or syntactic properties. The next section describes the design of a lexicon-driven pattern matcher that performs this sort of analysis prior to parsing, and the rest of the paper will present several types of examples where pre-processing serves to improve parsing.

LEXICO-SEMANTIC PATTERN MATCHING

The Pattern Language

Because the pattern matcher is designed as an efficient "trigger" mechanism and an aid in parsing, the patterns are mostly simple combinations of lexical categories. The patterns largely adopt the language of regular expressions, including the following terms and operators:

- Lexical features that can be tested in a pattern:
  - token "name" (e.g. "AK-47")
  - lexical category (e.g. "adj")
  - root (e.g. "shoot")
  - conceptual category (e.g. "human")
- Logical combination of lexical feature tests
  - OR, AND , and NOT
- Wild cards
  $ - 0 or 1 tokens
  * - 0 or more tokens
  + - 1 or more tokens
- Variable assignment from pattern components
  ?X =
- Grouping operators:
  <> for grouping
  [] for disjunctive grouping
- Repetition
  * - 0 or more + - 1 or more
- Range
  *N - 0 to N +N - 1 to N
- Optional Constituents
  {} - optional

The Rule Base

For the MUC-3 corpus, the knowledge base of patterns thus far contains about 150 rules, where each rule contains a pattern with an action (such as tagging, bracketing, deleting, adding, or otherwise enhancing the "tokenized" input to help the parser). The rules range from mundane combinations of words to intricate stylistic expressions. Below, we will go through some examples of some of these rules, and the next section will characterize their capabilities in more general terms. This is work in progress, so we will discuss both the current implementation and the directions for further work.

The strategy for pre-processing, as with parsing, is to process the text in stages, starting with coarse topic analysis and filtering, then moving on to tagging, segmentation, and template activation. Among the useful side benefits of the pattern matcher is that it discards portions of text that do not activate (or support) any templates. In MUC-3, this process eliminates about 75% of the input. On the first test set, the program did not skip any texts that contained relevant templates.

Because of this multi-stage design, the first stage of pattern matching contains the simplest patterns, and these include mostly expanded morphological forms, to avoid even the morphological analysis of large portions of irrelevant text. Below are three examples of these activator rules:

```
;;; rule 11
?PIVOT=(or found left shot) ?OBJ=* ?EFFECT=dead
       => (mark-activator murder d-vp) ;

;;; rule 40
?OBJ=$bombs ?ADJ=* ?PIVOT=(or shook exploded destroyed
                           destroying damaged damaging)
            => (mark-activator bombing b-s) ;
```

In addition to providing a rough screen of the input, these coarse template activation patterns "mark up" the text. Variable assignments effectively tag portions of text to help the parser. For example, the PIVOT tag tells the parser to favor a particular lexical term for the head of linguistic attachments, and the OBJ tag tells the semantic interpreter to try to fill a conceptual *object* role for a constituent. Since these patterns perform only the crudest form of linguistic analysis, their purpose is not to replace parsing but to allow the parser to focus its processing and not "prune off" paths that are likely to be critical.

Rule 11 above handles inputs such as *The attack left 9 people dead.* Rule 40 handles, for example, *The dynamite charge partially destroyed the bank facilities.*

The macros on the right hand sides of rules, such as *mark-activator*, generally use the results of the pattern match, including variable assignments, along with some other constants, such as *murder* and *d-vp*, to tag and segment the text. Template activation tags, like *murder*, allow the semantic interpreter to fill slots and apply constraints from the appropriate template during parsing. Grammatical tags, like *d-vp* (the double-object verb phrase, including adjectival complements) give a preferred parse, so the parser can try to favor a parse consistent with the lexico-semantic pattern.

The second set of rules, after the initial filtering and triggering, performs the cleanup of the input text, including many names, dates, punctuation, and marking of locative and temporal phrases. These rules can be somewhat more involved, as in the following examples:

```
;;; rule 97
?N=fullname
*comma*
?APP=<(not after fullname rpnoun aux
       past_part_verb pres_part_verb)
      (not fullname was were *comma*
       *semicolon*)* >
(or *semicolon* *comma*)
        => mark-appositive ;

;;; rule 113
{*comma*} ?PREP={(and prep (not between of))}
{det} $ ?TYPE=$loc+2 [of <known as>] {det}
[<?NAME=place-name ?TYPE={$loc}> ?NAME=$loc]
{*comma*}
?POSS={<*apostrophe-s* capital>}
        => mark-location ;
```

Rule 97 helps to distinguish appositive phrases from lists, relative clauses, and other constructions with internal punctuation. The parser handles many punctuated forms using grammar rules or meta-rules, but these can quickly get out of control. A simple example is *He is in charge of the investigations of the deaths of Guillermo Cano, director of the newspaper El Espectador, and Jaime Pardo Leal, the president of the Patriotic Union.*

Rule 113 catches many locative expressions.

The most complex patterns perform tagging and segmentation of grammatical constructions. While these are probably the most interesting and promising for the general control of parsing, we have only begun to encode them. The following are two examples:

```
;;; rule 127
(or l-number numword) ?OBJ=*4 killed
(or coordconj conj)
*2 (or l-number numword)
?SPOT=(or injured wounded)
       => mark-ellipsis ;

;;; rule 128
{aux} ?V=verb_leave (or l-number numword)
?FOBJ=*4 ?E1=(or injured dead wounded)
{<?C=(or coordconj conj)
     *2 (or l-number numword)
     ?SOBJ=*4 ?E2=(or injured wounded)>}
       => mark-left-dead ;
```

Rule 127 recognizes many cases of ellipsis involving death and injury, as in *Six people were killed and five wounded*, and rule 127 segments examples where the verb *leave* is used to express death and injury, as in *left 6 people dead.* These rules often overlap, as rule 128 overlaps with rule 11. The motivation for this is that rule 11 simply spots certain cases where *left* is used to express death (a very small percentage of occurrences of *left*), while the more powerful rule, 128, tries to segment the objects and effects.

The Algorithm

When the system loads the pattern-activation rules, it indexes each pattern by the lexical features (i.e. the words, lexical categories, roots and concepts) of each of its constituents, distinguishing those that require lexical analysis from the word-only rules. At run-time, the pattern matcher performs the following four operations:

1. It examines each input token (only) once for any features that index pattern tests.
2. Each satisfied pattern test "triggers" its enveloping rule. The satisfied pattern tests are cached so subsequent occurrences of the same input token avoid the feature examination.
3. After all input tokens have been examined, the program matches all triggered rules (those that have all of their non-optional tests satisfied) against the input. The matching uses a best-first search algorithm, where the "best" match is one that uses the most pattern constituents and the most

input tokens. This matching process is implemented as a table traversal.

4. The system executes the actions of all matched rules.

We now turn to how this simple form of pre-processing helps parsing and how it is likely to influence future advances in text interpretation.

FEATURES OF PRE-PROCESSING

This section gives some examples from news stories of the places where pattern matching eliminates or assists with work typically left for parsing. Pushing these tasks into this pre-processing phase with a less computation-intensive mechanism speeds up language analysis, reduces the complexity of the input texts, allows for modularity between topic analysis and data extraction, and increases the accuracy of the resulting analysis.

Pattern matching performs the following tasks:

1. Name recognition and reduction: Person names may contain long and complex titles and appositives, as in the following examples:

```
FORMER PERUVIAN DEFENSE MINISTER
GENERAL ENRIQUE LOPEZ ALBUJAR

MARIO SOLORZANO MARTINEZ, LEADER OF
GUATEMALA'S DEMOCRATIC SOCIALIST
PARTY,...
```

We recognize these constructs with the pattern matcher, using patterns that contain variables for first names and variables for titles.

2. Spatial phrase recognition and reduction: Pre-processing can easily identify and compress many locatives, using patterns that look for combinations of spatial prepositions with known locations, as in the following:

```
IN THE TOWN OF QUINCHIA, ABOUT 300 KM
WEST OF BOGOTA, IN THE COFFEE-GROWING
DEPARTMENT OF RISARALDA ...
```

3. Temporal phrase recognition and reduction: The pattern matcher picks out many temporal adverbial phrases, such as:

```
IN THE PAST FEW HOURS

MORE THAN 3 MONTHS AGO.
```

4. "Cleanup" of news style text: Patterns capture and help interpret style-specific constructions, as in the following examples:

```
OQUELI, 45,...

WITH LICENSE PLATE UF-2171
```

5. Tagging and segmentation: Some complex constructs, especially ellipsis and conjunction, are easier to identify and parse with identification at pre-processing, for example, the construct *<event>...<verb_leave>...Y <health_state> and...Z <health_state>* in the following sentence:

THE UPRISING, WHICH BEGAN AT 1100 (1700 GMT) ON 26 MARCH AND WHICH INCLUDES DEMANDS FOR BETTER JAIL CONDITIONS, HAS LEFT AT LEAST 12 DEAD AND SOME 20 INJURED, ACCORDING TO POLICE SPOKESMEN.

6. Topic analysis and filtering: Patterns for topical keywords and phrases help to perform topic analysis and filtering of stories. For example, stories containing patterns like <attack>...<civilian> are likely to be about terrorist attacks. This type of relevance determination is useful at the paragraph and sentence level as well. Eliminating irrelevant sentences saves the language analysis programs from having to parse them, and often avoids "false positives" by eliminating background information from interpretation, as in the following paragraph:

ALL OF THESE CHARACTERISTICS MAKE HONDURAS A DEMOCRACY, AND EVERY SECTOR OF HONDURAN SOCIETY SHOULD STRIVE TO STRENGTHEN THEM BY AVOIDING VIOLENCE AT ALL COSTS, OBEYING THE LAW, AND CONDEMNING AND ATTACKING EXTREME TERRORIST GROUPS REGARDLESS OF THEIR AFFILIATION, THUS ENABLING US TO CONSOLIDATE OUR DEMOCRACY AND REACH THE LEVEL OF POLITICAL DEVELOPMENT ATTAINED BY OTHER DEMOCRATIC COUNTRIES.

These six examples illustrate some of the places where fairly well-understood techniques from Artificial Intelligence, combined with a large lexical and conceptual hierarchy, are very useful in analyzing texts for natural language data extraction. In some cases, such as topic analysis, the pattern matcher operates as a separable component from the rest of the text processing system; in others, like syntactic segmentation and spatial/temporal reduction, it is more closely coupled with the parser. This approach has many of the advantages of phrasal parsing, such as robust coverage of a range of grammatical constructions, the elimination of grammatical complexity, and the easy adaptation of the system to handle sublanguage constructs. But it retains the advantages of parsing for handling agreement, attachment, and semantic interpretation of the text.

The next section compares this style of processing with earlier work in phrasal parsing.

PRE-PROCESSING AND PHRASAL PARSING

We have pointed out some of the problems with traditional left-to-right single pass parsing methods, including the lack of influence of global context on local interpretation, the complexity

of long sentences, stylized constructs, and garden paths. These well-known symptoms of syntactic parsing point to two general means of improving control—sublanguage analysis [2] and domain-driven or conceptual analysis [3]. Roughly speaking, this means that the system must constrain the input through either linguistic or semantic methods wherever possible. Pattern matching is an effective vehicle to enforce such constraints.

In some ways, the use of pattern matching for pre-processing is reminiscent of phrasal styles of language analysis [4, 5, 6], which in turn derive in part from semantic grammars [7]. These controlled styles of parsing were especially useful for engineering applications in limited domains, where it is much easier to cover the range of meaningful expressions and their interpretations than to control left-to-right parsing and subsequent semantic interpretation. However, phrasal analysis, like syntax-first parsing, tried to treat most of language interpretation within a single-pass, single-strategy process. This confined the approach to fairly simple applications and made it difficult to port from one domain to another.

In addition to this brittleness and limited scope of phrasal parsing, the phrasal approach suffered from a more fundamental problem: treating phrases or constructions as a replacement for grammatical rules seemed to miss the point of grammar entirely, leaving no place to account for most of the regularities of language. Even most of the rigid constructions and "idioms" of a language (like *riddled with bullets*) are grammatical. Thus the encoding of most of the knowledge about these expressions was really redundant, forcing the phrasal analyzer to apply interpretation rules and enforce constraints that easily could have been expressed in more general terms. This causes problems both in developing broad coverage and in applying automated methods of acquiring phrasal knowledge.

Lexico-semantic pre-processing, by introducing domain constraints and linguistic constructs prior to processing, controls parsing through two vehicles: (1) Triggering grammatical constructs prior to parsing allows the parser to apply the same grammatical knowledge to many different types of input without being constantly led to garden paths or false interpretations, and (2) Using filtering and template activation to capture some domain knowledge prior to parsing allows the parser to direct attachment and pruning toward the production of relations that affect the domain result. This sort of multi-stage analysis seems to be the right style for accomplishing the directed processing of the phrasal and sublanguage approaches while allowing for the breadth and portability that current text processing applications require.

CURRENT STATUS AND FUTURE ENHANCEMENTS

Our system, known as the GE NLToolset [8], is one of the more complete and mature text interpretation programs, having developed from a substantial research thrust into several applications outside of the research laboratory. Like other researchers in text interpretation, we have come to evaluate this sort of work in part through the system's performance on government-sponsored benchmark evaluations.

The second message understanding evaluation conference, in 1989, known as MUCK-II [9], used a corpus of slightly over 100 naval operations reports, with a final test on 5 such messages. The current MUC-3 development corpus contains 1300 open-source foreign news stories, with a final test on 100 such stories. The total corpus for MUC-3 is about 400,000 words, compared with about 3200 for MUCK-II. The MUC-3 task also requires both broader and deeper analysis of the texts, with an unconstrained range of responses. For example, the example sentence about the bombing in Risaralda would produce the template illustrated in Figure 1.

The MUC-3 evaluation scores each system on its ability to match a "correct" set of over 100 filled templates on 100 news stories.

The scale-up over the two years from MUCK-II to MUC-3 has strained parsing systems in throughput, coverage, and accuracy, and pre-processing has been essential to all three. Our system throughput is now an order of magnitude greater in words per minute (about 1000/minute) than in MUCK-II, the coverage is orders of magnitude greater, and the accuracy is about the same as at this point in MUCK-II, in spite of a harsher scoring system. Improvements in the grammar, lexicon, preference module, and recovery strategies have helped in this advance. However, large improvements in parsing are hard to come by, hence the incremental contribution of pre-processing is disproportionate, given the simplicity of the algorithm and rules.

Two major challenges remain in integrating the pattern matcher more effectively with the parser, and both should be accomplished, at least in part, before the end of MUC-3. We view the apparent success of simple pattern matching methods not as a replacement for real parsing, but rather as an example of how much work is involved in controlling parsing of texts. The current coupling of the parser with the pattern matcher is not sufficiently fluid to take advantage of much of the information that the pattern matcher can produce, leaving room for further integration.

The first apparent challenge is to tie the linguistic patterns, where appropriate, to "top-down" domain knowledge. In many cases, the common expressions, forms, and preferences derive from conceptual relationships in the domain; for example, the *leave dead* expressions are part of a general class of descriptions that follow events with the effects of events. For efficiency, the pattern matcher must recognize these descriptions at the lexical level, but there is no reason why domain knowledge cannot help to collect and create such lexical patterns.

The second challenge is to use the results of pattern matching more for parser preferences, using a strategy we call *relation-driven control*. This strategy looks for attachments of phrases to *pivots* which appear at the head of template activators. We have already implemented relation-driven control as a means of recovering from failed parses, but much of the task of using pivots and brackets to guide preferences remains.

In addition to these two challenges, another task, which is more difficult than it would seem, is to combine pattern matching with other, more syntactic methods of pre-processing, such as stochastic analysis or finite-state recognition of constituents.