

Grammars and Programming Languages:
To Further Narrow the Gap

Paula S. Newman
newmanp@acm.org

Proceedings of the GEAF 2007 Workshop

Tracy Holloway King and Emily M. Bender (Editors)

CSLI Studies in Computational Linguistics ONLINE

Ann Copestake (Series Editor)

2007

CSLI Publications

<http://csli-publications.stanford.edu/>

Abstract

Symbolic parser/grammar combinations can be viewed as programming systems for natural language processing applications. From this perspective, they can be compared with conventional programming systems, and seen to require more effort in the important development activities of testing and debugging. This paper describes tools associated with the RH (Retro-Hybrid) parser that facilitate these activities and are, to varying extents, more broadly applicable. The paper also suggests a new approach to improving parser efficiency using constrained inputs, based in part on one of the RH debugging tools.

1 Introduction¹

Developing general-purpose symbolic natural language parsers and grammars is difficult. One way of appreciating the difficulties is to view parser/grammar combinations as programming systems for natural language applications, and to compare them to traditional programming systems. From this perspective, it can be seen that far more effort must be devoted to testing grammars, reviewing test results, and debugging, than is required for traditional programs. Also, although considerable effort has been devoted to efficiency, parser execution tends to be slow, especially for deep-unification based grammars. Many innovative approaches have been applied to these problems. Copestake (2002) describes a comprehensive development system for HPSG grammars, and the XLE Online Documentation (2006) does the same for LFG grammars.

This paper explores some additional possibilities, based on tools developed for the relatively shallow RH parser (Newman, 2007a, 2007b), or suggested by those tools. The necessary background for the RH parser is provided in section 2. Then three problem areas are addressed. Section 3 discusses test/review problems, and broadly applicable methods of alleviating them using tools exploiting the TextTree display format (Newman, 2005). Section 4 discusses the difficulty of identifying failure points in debugging declarative grammars, shows how the difficulty is avoided in RH, and suggests some related approaches that are more widely relevant. Finally, section 5 suggests an approach to improving parser efficiency that adapts and combines: (a) a technique for leveling differences among parser outputs, for purposes of measurement (Ringger et al., 2004) with (b) methods of constrained execution, one of which is used in RH debugging

¹ Acknowledgments: Thanks are due to John Sowa, who made many helpful comments on an early draft of this paper, and also to an anonymous reviewer of another paper, who partially motivated this one by implying that developing deep-unification-based grammars is easy.

2 Background: The RH Parser

The RH (Retro-Hybrid) parser combines two major components, a shallow parser, and an overlay parser. These components are outlined below.

2.1 Shallow Parser

The shallow parser used in RH is the Xerox Incremental Parser (XIP), developed by Xerox Research Center Europe. XIP is actually a full parser, whose sentence output consists of a single tree of basic chunks, together with identification of (sometimes alternative) typed dependences among the chunk heads (Ait-Mokhtar et al. 2002, Gala 2004). But because the XIP dependency analysis for English was not mature at the time that work on the RH parser began, and because a classic parse tree annotated by syntactic functions can be more convenient for some applications, the overlay parser uses only the output chunks.

XIP is astonishingly fast, contributing very little to overall RH parse times (about 20%). It consists of the XIP engine, plus grammars for many languages. The grammar for a particular language consists of:

- (a) a finite-state lexicon that produces alternative part-of-speech and morphological analyses for each token, together with bit-expressed subcategorization and control features, and (some) semantic class features,
- (b) a substitutable tagger that identifies the most probable part of speech for each token, and
- (c) sequentially applied rule sets that extend and modify lexical analyses, disambiguate tags, identify named entities and other multiwords, produce basic output chunks, and identify inter-chunk head dependences. (Note: the dependency rule results are not used in the RH hybrid.)

2.2 Overlay Parser

The overlay parser uses a guiding grammar expressed as a collection of networks that are similar to Augmented Transition Networks (Woods, 1970), thus the term "retro". A recursive control mechanism traverses the grammar networks top-down and depth-first to build constituents.

The grammar network arcs are labeled by references to tests for specialized categories. These tests, if successful, either return a shallow parser chunk or a parse forest, with the latter obtained by recursively invoking the control to traverse another grammar network. The specialized category tests are context-free, and, if performed, their results are cached. But the tests are often gated by pre-tests referring to contextual considerations such as parent category and left-sibling features. An extensive preference scoring system (see Newman, 2007b)

is used to prune partial parses early, and to select a single "best" parse, with scoring ties resolved by low attachment considerations.

3 Grammar Test and Review

3.1 The Problem

Conventional applications based on traditional programming languages are built using a more-or-less standard development process that assumes applications consist of relatively independent, modular units. As an application is specified, written, and unit-tested, a collection of global tests reflecting meaningfully different input combinations is constructed. Before the application is released, the tests are executed iteratively, and errors are removed, until the results are correct. In general, the number of input combinations that must be tested on a global level is relatively small, and determining whether results are correct is straightforward.

In contrast, for symbolic NL grammars, the test/review process is an integral part of grammar construction, and involves applying the grammars as a whole to as many input examples as feasible. This is because grammar elements are not modular in the same way as conventional programs, and the number of meaningfully different input combinations for a target genre is usually enormous.

Furthermore, determining the correctness of test results is far more difficult than for conventional applications. A standard result representation is a parse tree rendered in node + edge form. This representation, although useful for short sentences, is not easily scanned for longer sentences, which are very frequent. For example, the many non-fiction documents (in several genres) that were used to refine the RH English grammar have an average sentence length of roughly 20 words, with a standard deviation of about 11. Thus many parse trees obtained by parsing those documents are twenty or more words wide at the leaves, plus intervening blanks, so that the trees do not fit into a single window. Parse trees can also be very deep, making it difficult to grasp the structure.

3.2 TextTrees

TextTrees were developed for RH grammar test and review, and can also substantially reduce test/review effort for other types of syntactic grammars. TextTrees are linearly rendered, flattened, parse trees that convey right-side dependency by indentation. They can be read as prose, but at the same time expose most parsing errors.

An excerpt from a TextTree display obtained by parsing section J42 of the Brown Corpus is given in Figure 1. The indentations of the first TextTree show

4 (2) Thus the Congress marks a formal recognition of the political system that was central to world politics for a century. [best more morett chunks](#)

```
Thus
the Congress
marks
  a formal recognition
    of the political system
      {that
        was
          central
            to world politics
              for a century}.
```

5 (8) International law had to fit the conditions of Europe , and nothing that could not fit this system , or the interests of the great European nations collectively , could possibly emerge as law in any meaningful sense. [best more morett chunks](#)

```
International law
had to fit
  the conditions
    of
      [Europe,
        and
        nothing
          {that
            |could not| fit}]
          {[this system,
            or
            the interests
              of the great European nations]
            collectively,
            could possibly emerge
              as law
                in any meaningful sense}.
```

Figure 1. Example TextTrees

a correct structure, but those of the second show a coordination of "Europe" and "nothing" as the object of the preposition "of". The probable correctness of the first sentence and the errors of the second sentence can be seen quickly, even though the sentences contain 20 and 35 words respectively.

The TextTree construction algorithm is given by Newman (2005). The algorithm is almost parser-independent, requiring only a <category, style-name> pair

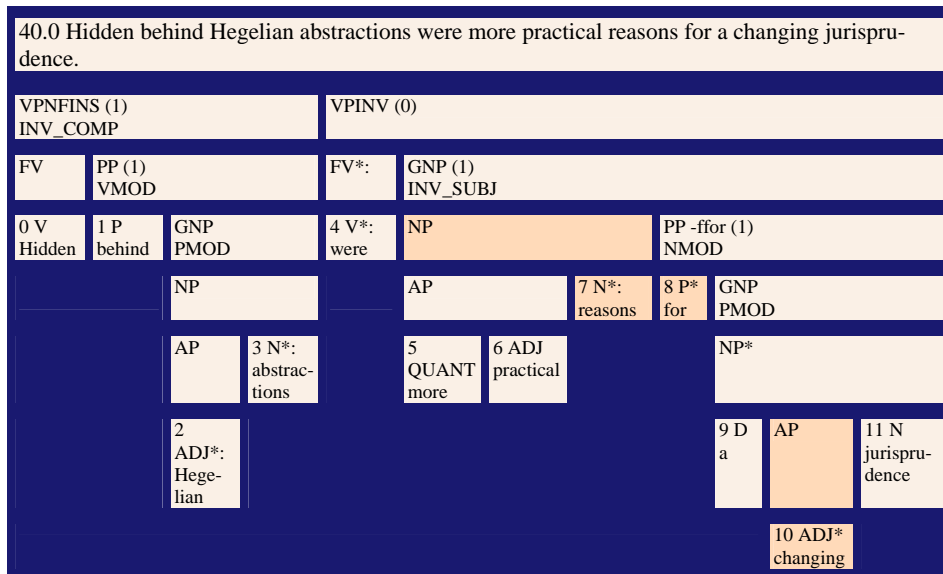


Figure 2. A full parse tree

for each constituent category type used by the parser. Limited empirical tests (see reference) show review speed improvements of 25% to 50% relative to using full parse trees, depending on the effort made to obtain a high review speed. Also, TextTree display files provide a good grasp of overall document problems.

3.3 Other Uses of TextTrees

TextTree displays, like those of Figure 1, have been the primary means of reviewing RH parser results, allowing rapid determination of whether parses identified as "best" are probably correct, and providing access to other results. Each sentence in the display has the following links:

- a) The **best** link leads to a display of the full parse tree² for the best parse. Figure 2 gives an example. We note that the Figure 2 tree is for a sentence of only 12 words, yet spans the page in this format, even with some tokens split between lines. Although full-screen presentations accommodate wider trees, trees for long sentences must be split into segments, arranged vertically.

² The full parse tree displays are in a TreeTable format (Newman, 2002). Parent nodes physically span children, giving a good appreciation of structure. For applications with narrower trees, the cells can contain indicative content. In the parse tree adaptation, node features are accessed by mouse-over of category names.

- b) The **chunk** link leads to a display of shallow parser output in the same format as for full parse trees, with basic chunks shown as children of the (omitted) top node.
- c) The **more** and **morett** links lead to displays of other parses retained, in full parse tree form and TextTree form, respectively. The number of parses retained depends on parser invocation options, outlined below.

```

0. International law had to fit the conditions of Europe, and nothing that could
not fit this system, or the interests of the great European nations could possibly
emerge. 0.40 (s = 4) 0.35 (s = 4)
International law
had to fit
  the conditions
  of
    [Europe,
    and
    nothing
      {that
        |could not| fit}}
      {[this system,
        or
        the interests
          of the great European nations]
        could possibly emerge}.

0. International law had to fit the conditions of Europe, and nothing that could
not fit this system , or the interests of the great European nations could possibly
emerge .0.4 (s = 3)
[ {International law
  had to fit
    the conditions
      of Europe},
and
{nothing
  {that
    |could not| fit
    [this system,
    or
    the interests]
    of the great European nations}
  could possibly emerge.} ]

```

Figure 3. Excerpt from **morett** display

The number of trees retained by the parser depends on the parser invocation mode. In "production" mode, only the best parse is retained, so the **more** and **morett** links are not of interest. In "standard" mode, all parses with the highest preference score are retained, while in "nopruner" mode preference-based pruning is disabled, and many more parses may be completed and retained.

The **morett** link supplies additional TextTree-based information. Figure 3 excerpts a **morett** display of "nopruner" mode results for a reduced version of the incorrectly parsed sentence of Figure 1. Duplicate structures are grouped. Figure 3 first shows a strange, incorrect structure, reflected in two parses, both given the preference score $s = 4$, and then a better structure, with score $s = 3$, indicating that one source of the problem is in the preference assignments. (The correct structure appears slightly further on in the display.) If the display were produced in standard mode, only the highest scoring parses would be shown.

The **morett** display thus serves several purposes. First, it highlights probably duplicate parses. Also, when pruning is enabled, it shows parses that were retained when they should not have been. Finally, by grouping duplicate structures (thus limiting the number of structures to be scanned), it facilitates determining whether the correct parse was found, even if not identified as best³

A further RH use of TextTrees (not illustrated) allows a more direct determination of whether a correct structure was obtained. A "find" parser option accepts single sentences in TextTree format and produces as output a list of matches found in the retained parses, with links to the full parse trees.

All these TextTree-based RH tools should be useful in other parser development environments. Another potential use beyond those discussed is in regression testing, to illustrate changes in results. But a caveat is needed. TextTrees are clearly useful for SVO languages with mostly projective grammars (i.e., with mostly contiguous constituents). Their adaptability to other types of languages has not been explored.

4 Grammar Debugging

4.1 The problem

Debugging erroneous grammar outputs addresses several questions: why a correct parse was not found, why some incorrect parses were found, and why duplicate parses were found.

³ The number of trees shown via the **more** and **morett** links is somewhat constrained, to limit the size of files containing full parse trees. Currently, if only one sentence is being parsed, the highest scoring 100 trees are shown; otherwise a random sample of 30 is used.

In comparison with conventional applications, grammar debugging involves more effort, partly because it is far more frequent, being an integral part of the continuous test/review process. But the major reason for the increased difficulty is that traditional debugging approaches, which rely to a large extent on tracing program execution, are usually not applicable. The tracing can involve obtaining application specific intermediate results at critical points in the program, and/or using language-processor-supplied debugging tools to step through code related to errors. Analogous tracing of grammar "execution" would follow the application of the grammar rules. However, useful traces require the existence of a simple, accurate mental model of the execution sequence and, unfortunately, parsers rarely conform to such a model. As an important example, chart parsing makes the sequence of parser operations difficult to predict and follow, because adding an edge to the chart can activate the continuation of many rules to which that edge might be relevant. Also, general traces of rule application are likely to be extremely voluminous, because of the number of alternatives examined.

Therefore, debugging tools for grammars tend to focus on accumulated results. For bottom up parsing, displays can be constructed to illustrate the successive node attachments that were made, linked to the associated grammar rules, to aid in detecting both inappropriate and missing attachments. Such displays, however, can be quite complex, even for relatively short sentences. Copestake (2002) discusses how the complexity can be reduced, by highlighting nodes which are ancestors or descendants of an interactively selected node.

For top-down parsing, creating equally informative post-parse displays is more challenging. A bottom-up display of attachments can only contain constituents that were eventually expanded down to tokens, and so are of less help in discovering missing rules or rule components.

In the next subsection we discuss why and how traces are used successfully in RH grammar development. A subsequent subsection discusses an extension which is more generally applicable.

4.2 RH debugging with traces

While, for many parsers, tracing is not practical for grammar debugging, it is useful for debugging an RH overlay parser grammar, for three reasons:

1. Most important, overlay parser execution is a simple, top-down/depth-first process, and therefore can be traced by a hierarchically indented sequence recording significant parser actions.

2. Trace volume is limited, because the overlay parser operates on disambiguated tags and chunks.⁴
3. The ATN-like grammar networks provide symbolic information that can be replicated in the traces. An excerpt of a grammar network is shown in Figure 4, with each row representing one or more network arcs in terms of a single **From** state, possibly multiple **To** states, and a **Label** of a specialized category test. (For a fuller description of the networks, see (Newman, 2007a)). The relevant aspect here is that searching and following the traces is aided by the inclusion of the network names, row information, as well as general category names within the trace output. Figure 5 shows an excerpt from an overlay parser trace that is attempting to develop a coordinated NP (category NPC) using the network of Figure 4, starting at token position 0. The trace shows a test for pre-coordinators, and then goes on to a test for a simple (non-coordinated) noun phrase. (The excerpt is edited to remove detail and expand some abbreviations).

From	To	Label	Other material & comments
NC_1	NC_2	T_PRE_COORD	// e.g., both, either, between
NC_1	-1	T_CUT	// meta-test, like prolog cut // stops testing if prior test satisfied
NC_1	NC_3	T_NPS	// simple noun phrase
... Omitted...			
// After first simple noun phrase			
NC_3	NC_5	T_COMMA	
...Omitted...			

Figure 4. Excerpt from NPC_NET for parsing coordinated noun phrase

<ul style="list-style-type: none"> • Parsing NPC:0 in NPC_NET entry NC_1 • Continue NPC(0:0) NPC_NET at NC_1, test T_PRE_COORD <ul style="list-style-type: none"> • do_T_PRE_COORD:0 • do_T_PRE_COORD:0 : failed • Continue NPC(0:0) NPC_NET at NC_1, test T_CUT • Continue NPC(0:0) NPC_NET at NC_1, T_NPS <ul style="list-style-type: none"> • do_T_NPS:0 • Parsing GNP:0 in NPS_NET entry N_1 • Continue GNP(0:0) NPS_NET at N_1, test T_TITLE
--

Figure 5. Excerpt from trace for a coordinated noun phrase at token position 0

⁴ Erroneous RH parses that are the result of errors in tagging and chunking are found via the **chunks** display, and can be investigated using the very helpful XIP trace facilities.

4.3 Debugging with Constraining Inputs

A more generally applicable approach to grammar debugging uses constraining inputs, which are sentences with some substrings bracketed and category-labeled, for example "{NP: *Time*} *flies* {PP: *like an arrow*}.}

Constraining inputs can be applied in different ways, depending on parser direction and type. For bottom-up parsers, the constraining inputs can be used to identify points of failure by limiting a bottom-up chart to constituents that are bottom-up consistent with the brackets and labels. For top-down chart parsers, the constraining inputs might dictate a parse of each bracketed element, starting with the lowest. A failure would occur if there is no parse for a bracketed element, or if no parse for a bracketed element includes all its contained bracketed elements in the input.⁵

For the RH overlay parser, constraining inputs are used to restrict parser execution. This reduces trace volume and limits the effects of an inconvenience introduced by caching, namely that all but the first invocation of a test at a token position returns a cached result, and the trace indicates only either failure or a success. In that case, the trace must be searched from the beginning to find the trace of that first invocation.

However, only simple bracketings have been used in RH debugging, because effective top-down bracketing requires that the brackets serve as barriers. Any bracket beginning at a token position p must be preceded by any higher level brackets also beginning at p . Figure 6 shows effective and ineffective bracketings to constrain processing of a sentence to a declarative, rather than imperative, interpretation with an initial NP.

The constraining brackets need not be precise with respect to punctuation enclosure. Methods for allowing this are described further on, in the context of using constraining inputs to improve parser performance.

Summarizing the debugging tools discussed above, literal traces of parser activity are used directly for RH debugging, and their convenience is improved by the use of constraining inputs. While tracing parser activities may not be suitable to other parser environments, constraining inputs may well be helpful in other ways to aid in debugging.

⁵A somewhat related facility is provided by XLE (XLE Online Documentation, 2006). It allows the complete trees retained after the parse to be searched for ones consistent with a bracketing. The facility might thus be used to find a point of failure by applying the facility multiple times, each time adding higher brackets

<i>ANY: {NP: You} love Mary}}</i>	will not rule out imperative reading
<i>{ROOT: {NP: You} love Mary}}</i>	will rule it out
<i>{NP: You} love Mary</i>	parse fails because <i>NP</i> not outermost at 0

Figure 6. Constraining inputs

5 Parser Efficiency

5.1 The Problem

Deep-unification-based parsers do not currently approach the efficiency of the fastest parsers. Figure 7 gives approximate relative parse times of several parsers for the Penn TreeBank Wall Street Journal Section 23. The relative times are derived from reports comparing the efficiency of Collins Model 3 (Collins, 1999) with one of the other parsers, when executed on the same machine.⁶

The relative times for the XLE LFG parser, considered a very fast unification-based parser, are based on a report by Kaplan et al. (2004). Results are given for both core (reduced) and full English grammars. The relative time for the fast stochastic parser by Sagae and Lavie (2005) is derived from that reference, and that for the RH parser is based on results reported by Newman (2007a).

Deep-unification-based parsers tend to be slower because unification is a destructive operation that requires copying of the structures to be unified, using large amounts of time and space. Sophisticated methods have been developed to limit this cost (Maxwell and Kaplan, 1996), but do not remove the gap.

5.2 Current Approaches

An important current approach to the efficiency problem uses the results of fast partial parsers to constrain, or establish preferences for, follow-on parser search paths (Frank et al, 2003 and Daum et al., 2003).

	Time
Sagae & Lavie 2005	.25
RH 2007	.31
Collins model 3	1
XLE/LFG core English grammar 2004	1.5
XLE/LFG complete English grammar 2004	5

Figure 7. Relative time comparison

⁶ We do not include relative times for the probably faster stochastic CCG parser by Clark and Curran (2007), because their comparisons are explicitly stated to be indicative only, in that the CCG results were obtained on a faster machine than the other parser results.

<p>{N'' {SPEC the} {N' {N boy} {P' ... }}}</p> <p>{NP {NP {DET the} {N boy}} {PP ... } }</p>
--

Figure 8. Two parses for "the boy in the park"

It should be possible to pursue this direction further, that is, to constrain deep, relatively slow, parsers by the results of full, fast, parsers. The difficulty with doing so is that the results of an arbitrary front-end parser are unlikely to be consistent with those of a back-end parser in terms of constituent structure, labeling, and punctuation enclosure. For example, different parsers might deliver the different structures shown in Figure 8 for the same noun phrase.

Cahill et al (2007) describe one way of addressing this difficulty. They train a fast statistical parser on the uncorrected outputs of the target back-end parser for a large corpus. The resulting trained parser then serves as the constraining front end. This removes the inconsistency problem, at the price of not training on a gold standard. It can thus improve performance and coverage (the latter because the constraints can prevent the parser from exceeding imposed resource limits), but the potential improvements in accuracy that might be obtained by training on fully-correct parses are not realized. The method is reported to obtain speedups of approximately 1/3.

5.3 Alternative Possibility: Leveling Differences

Another way of using a faster parser to constrain the execution of a slower, deeper one is to adapt an approach developed by Ringger et al (2004). That approach is intended to remove differences between parse trees for purposes of measurement, for example, to compare the correctness of parser results against a treebank. It focuses on removing brackets from non-maximal projections of heads, and essentially consists of the following steps:

1. General transformation rules developed for the pair of gold standard trees and the output trees of the parser to be measured are applied. Most transformation rules just modify the identification of phrasal heads.
2. Then, brackets representing non-maximal projections of heads are removed from both sets of trees. After this operation the result for the first parse of Figure 8 would be simply: {N'' the boy {P'' ... } }
3. Finally, after other (not specified) pair-specialized transformations are performed, the resulting trees are compared using unlabeled bracketing.

To adapt the approach for purposes of constraining parser execution, transformation rules would necessarily be restricted to the outputs of the fast front-end parser. Then brackets and labels would be retained only for maximal projections of heads.

The resulting transformed results of the front-end parser could be used in different ways to constrain a back-end parser, depending on the back-end parser approach. Two examples are given.

Top-down usage. For top-down parsing, the approach sketched in the preceding section on debugging the RH overlay parser using constraining inputs is applicable. Bracketing by maximal projections of heads (with heads identified by the front-end parser adjusted if necessary) should satisfy the requirement that the brackets serve as barriers, that is, that any bracket beginning at a token position p must be preceded by any higher level brackets also beginning at p . To deal with differences in category labeling and punctuation enclosure, the following method is used:

1. Opening brackets in the input are considered to extend over an interval $\langle b', b \rangle$, where b is the actual position of the bracket, and b' is the first of possibly several punctuation tokens immediately preceding b .
2. Each back-end parser category is considered equivalent to ANY of the constraining categories which it might match.
3. In parsing a constituent, when the parser reaches token position p within the next open bracket interval $\langle b', b \rangle$, and that bracket has category c , the parser will not process a test for a category sc unless $sc = c$, or sc is considered equivalent to c , or sc is a punctuation test.
4. When processing a constituent corresponding to a bracket pair ending at a position eb , no tests are processed for the constituent beyond eb except for immediately following sequences of punctuation.

Bottom-up usage. For bottom-up head-driven parsing, the constraining structure might be used in roughly the following way:

1. Restricting token tags to those given by the constraining parse (suitably mapped)
2. Initiating a constituent c using a lexical head h only if there is a constraining subtree containing h that is headed by h .
3. When a constituent c with lexical head h is to be extended by some dependent d , the extension is accepted only if there is a lowest constraining subtree $LCTc$ larger than c that also has lexical head h , and d is either a token or equal to another constraining subtree, and $LCTc$ includes d .

In either usage case, if no parse is obtained by the combination, the deeper parser might be used directly.

These approaches would also address the ambiguity problem, limiting the burden on current methods of supplementing unification-based grammars with stochastic, corpus-based post-processors for disambiguation (Kaplan et al. 2004, and Toutanova et al. 2005). (The suggested approaches do not replace post-processing, because a particular syntactic structure might have multiple associated deep structures requiring disambiguation.)

5.4 Experiment

A small experiment was performed to test the potential performance improvement for the RH overlay parser using the method combination for top-down parsers described in the previous subsection. Specifically, we combined a simulated result of the described adaptation of the Ringer et al. (2006) approach, with the RH approach to debugging using constraining inputs.

For a sentence that required an inordinate amount of parse time, the overlay parser was constrained by a manually constructed bracketed input, shown in Figure 9, with the brackets used only for maximal projections of heads.⁷ The results were not encouraging, because even for a sentence that the parser found difficult, the performance improvement was only about 35%.

However, the approach should realize more significant gains if used with a slower parser, because: (a) the RH overlay parser builds on the results of the fast front-end shallow parser, so no time is spent in considering alternative token tags or basic attachments, and (b) the work performed by the overlay parser for each potential constituent is far less than that performed, for example, by unification-based parsers.

6 Summary

We have described some development tools associated with the RH parser, both mainstays, and some more recent extensions. These tools bring the grammar development process closer to that of applications built using traditional programming languages, in the sense that they reduce the amount of effort required for result review and debugging. Many of the tools are relevant across grammar frameworks. We have also described an approach to improving efficiency partially suggested by one of the RH development tools. Figure 10 summarizes the existing and potential tools discussed, their usage in RH development, and their wider relevance

⁷ We note that the illustration is formatted just for readability. It is not a TextTree because it contains many brackets and labels, and the first PP is not indented.

```

{ROOT
{GNP It}
reached
  {GNP its ultimate philosophical statement}
{PP in {GNP notions
  {PP of `` {GNP state will "
    {VPNFINS put forward {PP by {GNP the Germans}}
    {PP especially by {GNP Hegel,}}}}}}}}
{SUBCL although
  {S
  {GNP political philosophers}
  will recognize
  {GNP its origins
  {PP in
    {GNP the rejected doctrines
    {PP of {GNP Hobbes}}}}}}}} .

```

Figure 9. A constraining input using maximal projections of heads

The most important of the tools described, TextTrees, allow most erroneous parser results to be rapidly identified. Also, displaying all retained parses in TextTree form assists in finding parses that were obtained, but were not identified as "best", and in identifying duplicates for further study and removal. TextTree displays are relevant to reviewing parser results for SVO languages with mostly projective grammars. Their relevance to other types of grammars is yet to be studied.

Parser execution traces have served as the primary tool for debugging the RH overlay parser grammar. A recent extension constrains parser execution by partially labeled and bracketed inputs, to make following traces more convenient. While execution traces are of questionable relevance to chart parsers, using constraining inputs in debugging are of wider relevance.

The paper also suggests a technique for using the results of a fast parser to constrain the activities of a slower one. The technique combines: (a) an adaptation of a method for leveling differences between parser results with (b) different methods of constraining the activities of a back-end slower parser, depending on parser approach. While the technique is not very promising for the RH hybrid, because the overlay parser builds on the results of a very fast shallow parser, it may provide significant performance gains for deep-unification-based parsers

Tool Area	Usage in RH development	Relevance beyond RH?
Test/Review		
Best texttrees	heavy	yes
All texttrees	some (relatively new)	yes
Find texttree in outputs	new	yes
Debug		
Via tracing	heavy	limited
Using constraining input	limited (relatively new)	yes
Efficiency		
Using constraining input	no	probably

Figure 10. Summary of tools, usage in RH, and applicability

References

- Cahill, Aoife, King, Tracy Holloway, Maxwell, John T. 2007. Pruning the Search Space of a Hand-Crafted Parsing System with a Probabilistic Parser. *Proceedings of the Workshop on Deep Linguistic Processing*, pages 65-72
- Clark, Stephen and Curran, James R 2007. Wide-Coverage Efficient Statistical Parsing with CCG and Log-Linear Models. To appear in *Computational Linguistics* 33(4). Draft at www.cs.usyd.edu.au/~james/pubs/pdf/cl07parser.pdf
- Collins, Michael. 1999. Head-Driven Statistical Models for Natural Language Parsing. Ph.D. thesis, University of Pennsylvania.
- Copestake, Ann. 2002, *Implementing Typed Feature Structure Grammars*, CSLI Publications
- Daum, Michael, Foth, Kilian A., and Menzel, Wolfgang. 2003. Constraint Based Integration of Deep and Shallow Parsing Techniques. In *Proc. 11th Conf. of the European ACL*, pages 99-106
- Frank, Anette, Becker, Markus, Crysmann, Berthold, Kiefer, Bernd, and Schaefer, Ulrich. 2003. Integrated Shallow and Deep Parsing: TopP Meets HPSG. In *Proc 41st Annual Meeting of the Association for Computational Linguistics*
- Kaplan, Ronald M., Riezler, Stephan, King, Tracy H., Maxwell, John T., Vasserman, Alex, and Crouch, Richard. 2004. Speed and accuracy in shallow and deep stochastic parsing. In *Proc Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 97-104

- Maxwell, John T., Kaplan, Ronald M. 1996. Unification-based parsers that automatically take advantage of context freeness. *Proc First Annual LFG Conference*
- Newman, Paula S. 2002. Exploring discussion lists: steps and directions. In *Proc Second Joint ACM/IEEE-CS Conference on Digital Libraries*, pages 126-134
- Newman, Paula S. 2005. TextTree Construction for Parser and Grammar Development. In *Proc Workshop on Software at 43rd Annual Meeting of the Association for Computational Linguistics*.
Available at <http://www.cs.columbia.edu/nlp/acl05soft/>
- Newman, Paula S. 2007a. RH: A Retro-Hybrid Parser. In *Proc Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume*, pages 121-124
- Newman, Paula S. 2007b. Symbolic Preference Using Simple Scoring. In *Proc 10th International Workshop on Parsing Technology*, pages 83-92
- Ringger, Eric K., Moore, Robert C., Vanderwende, Lucy, Suzuki, Hisami, and Charniak, Eugene. Using the Penn Treebank to Evaluate Non-Treebank Parsers, In *Proc 2004 Language Resources and Evaluation Conference*, pages 867-870
- Sagae, Kenji, and Lavie, Alon. 2005. A classifier-based parser with linear runtime complexity. In *Proc. 9th Int'l Workshop on Parsing Technologies*.
- Toutanova, Kristina, Manning, Christopher D., Flickinger, Dan, and Oepen, Stephan, 2005. Stochastic HPSG Parse Disambiguation using the Redwoods Corpus. *Research on Language and Computation* 3(1), pages 83-106
- XLE Online Documentation. 2006.
Available at <http://www2.parc.com/isl/groups/nlft/xle/doc/xle.html#SEC>