Pacific Association for Computational Linguistics

JAVA PARSER FOR HPSGS: WHY AND HOW

VLADO KEŠELJ

Computer Science Department, University of Waterloo, Waterloo, ON N2L 3G1, Canada

The design and implementation of a Java parser for Head-Driven Phrase Structure Grammars (HPSGs) is described.

This experimental parser is developed in order to further investigate the use of the HPSG model in the area of Internet information retrieval (IR).

The motivation for the parser and the main design issues, including implementation of a unification mechanism and chart-parsing in Java, are discussed.

Key words: HPSG, parsers, Java, unification, chart parsing, Internet information retrieval.

1. INTRODUCTION

The HPSG formalism for NLP is a high-level unification approach suitable for studies in theoretical linguistics and strongly influenced by Prolog. HPSGs also provide a very flexible model, in which a large number of NL phenomena can be effectively parsed, and whose grammars are robust and can be relatively easily managed. For this reason, the HPSG model is attractive for lower-level, practical applications, i.e., for emerging area of *natural language engineering.*

This attraction naturally leads to the problem of developing HPSG parsers in a low-level programming language. There has been some work reported on the problem (Makino 1998); however, we are not aware of any reports about finished systems implemented in Java.

For several reasons, Java is an especially attractive environment for the implementation of an HPSG parser. The similarity between hierarchical HPSG type hierarchy and OOP (object-oriented programming) hierarchy of classes can be conveniently used in parser and grammar design. Java is a good prototyping language, compared to C++ for example, easy to experiment with various approaches, which makes this shift in programming language paradigm less drastic and easier to implement.

More specific motivational arguments for an HPSG parser in Java include applications in Internet IR and some additional research questions in NLP.

Typical systems for Internet IR are search engines with keyword-based retrieval methods (e.g., boolean queries, vector space model, probabilistic models). For a long time, it has been recognized that this approach has an inherent limitation in precision and recall and various improvements and other approaches have been proposed. Many improved methods include NLP to a certain level. One of the drawbacks to this approach when applied to the Internet IR is that a large running time is needed for the task, compared to keyword-based approaches. CPU time is precious on a heavy-loaded IR server (e.g., a search engine), so this limitation puts tight restrictions on the extent to which NLP technology can be used. A solution to this problem is to incorporate an NLP module in an applet which is sent to the client side as a part of the search engine interface (typically an HTML page). A client machine is much less loaded, so NL processing on the client machine has more CPU time per unit of real time. When an NL query is processed to a keyword-based query on the client, the keywords are sent to the server.

For this approach, the applet has to be implemented in a portable, secure, and still relatively efficient programming language—making Java a good choice.

[1]This work is supported by NSERC.

Use of NLP in IR is not limited to the retrieval phase; it can be used in the indexing phase of advanced indexing methods, such as the methods for conceptual indexing. Mobile agents and multi-agent systems are suitable for this task, and they are most easily implemented in Java, as well.

The above client-side processing approach and Internet environment in general give rise to an interesting problem, which we call *distributed NLP.* Namely, an NL parser is inherently a large application. The lexicon alone must typically include hundreds of thousands of lexeme descriptions in order to have a sufficiently wide coverage of an NL and produce satisfactory parses. In suggesting that an NLP-capable applet is sent to a client, an obvious drawback with a direct approach is that an unacceptable amount of time is needed to transfer the applet over the network. Its time cost puts some doubt on our initial time-cost-based motivation for introducing the client-side NLP in the first place.

A solution is to assume that the parser is downloaded by the client before the retrieval time. This can be a perfectly satisfactory solution; however, there is another approach that offers more flexibility and a potential use in various other applications. If the parser is modular and functional without all of its modules, then only an essential, kernel part would be sent over the network, thus reducing the time and bandwidth required. For example, the lexicon can be divided in parts according to semantic domains. Initially, a kernel consisting of the most frequently used words, phrases, and rules is sent to the client, and it is sufficient to make some first parsing steps with the first query. When a query is issued, and afterwards, when a dialogue has started, various parts of language knowledge are retrieved from the server according to the keyword and semantic relevance as the dialogue focus is more precisely determined.

For this approach, a programming environment which supports dynamic code loading would be very helpful, if not required. Using two Java features—dynamic class loading and object serialization—it was possible to achieve the grammar and lexicon modularity in an elegant way.

2. BACKGROUND

In the construction of experimental HPSG grammars, we rely on the HPSG formalism as presented in Sag (1998). The ALE system (Carpenter 1992) is used to experiment with grammars, prior to using them in our Java parser.

The first HPSG chart parser was introduced by Proudian and Pollard in 1985. The parser was implemented in Prolog. In 1990, Popowich and Vogel described their HPSG chart parser, also developed in Prolog, which had a better coverage of the developments in the HPSG formalism up to that time. Many solutions and results presented in those two papers are reused in the development of this chart parser. In Carpenter 1999, a chart parser for probabilistic context-free grammars, implemented in Java is described. Malouf 1999 treats some efficiency issues regarding graph unification for parsing constraint-based grammars.

For the output semantic representation, the SystemX (Cercone 1993, Hall 1998) logical form is used.

3. UNIFICATION

Unification is one of the crucial concepts in the HPSG model. It is a built-in Prolog feature. Java does not include it, so a Java package is implemented in order to support it. The problem of implementing unification in a lower-level language is discussed in Perelman-

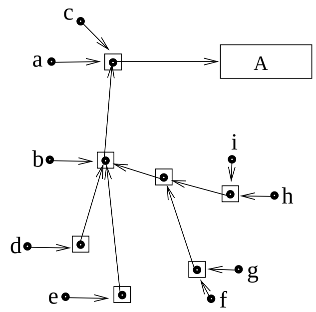

FIGURE 1. Unified variables

Hall 1995, where a C++ implementation is described. The same basic idea is used in our parser; however, due to differences between C++ and Java, and specific requirements of our unification package, our implementation is significantly different.

In the center of our unification package, the class *Unifiable* is defined. The class Unifiable is an abstract class, and all objects that can be unified are instances of its subclasses. There are four subclasses, which are needed to properly and efficiently implement the HPSG model. These classes represent atoms, attribute-value matrices, types, and arrays. The package is built as a part of the parser, but it is a general graph-unification package and can be used for other tasks.

Internally, unifiable objects are accessed through an auxiliary class called *UnifPointer.* The class simulates pointers, which are not present in Java. As an illustration, Figure 1 shows a data structure in which variables a, b, …, i are unified and being instantiated with the same object A. The path compression technique, like in union-find data structure, is used to reduce the cost of tracing the pointers.

The main unifiable class for our problem is ATM (attribute-value matrix). Several unification procedures are implemented, including destructive and non-destructive procedures.

4. PARSING

With the unification mechanism just described, we now give a description of the chart-parsing algorithm.

In a typical chart-parsing algorithm, a chart is filled from left to right and the procedure is expectation-driven; i.e., we know that the start symbol is the parsing goal, and using the rules we direct our search in that direction. In the parsing process, the *dotted rules* (as in the Early's algorithm, e.g., NP $\rightarrow$ THE $\cdot$ ADJ N) are used.

There are several issues that have to be handled differently in the HPSG chart parsing than how it is typically done.

4.1. Space is not reused.

Although we do reuse time by using the chart, we cannot reuse the space in the HPSG chart parsing. E.g., let us consider the following HPSG rule:

$$\mathrm{S} \rightarrow \begin{bmatrix} \textit{phrase} & \\ \mathrm{HEAD} & \begin{bmatrix} \textit{noun} & \\ \mathrm{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix} \begin{bmatrix} \textit{phrase} & \\ \mathrm{HEAD} & \begin{bmatrix} \textit{verb} & \\ \mathrm{AGR} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

If we apply this rule to two entries, parts of them will become unified (the values of the attribute AGR), so they will not be independent any more, and cannot participate in other parent constituents. For this reason, we cannot include daughters in their mother simply by referring to them, we must clone the daughters and then include them in the new mother entry.

4.2. Island parsing instead of left-to-right expectation-driven parsing

The left-to-right expectation-driven parsing is not used—the island parsing is used instead. In island parsing, the parsing starts from each word in the sentence and goes in both directions. The algorithm can be described in the following way: Each word in the sentence becomes an active entity which creates complex components according to given rules, being the head of those components. Not all HPSG rules have heads, but in those cases we can choose a dummy head—the left-most component being the best candidate.

The above short description is related to an implementation idea based on creating an active agent for each word in the sentence. It may be debatable whether we can call such entities agents, but in any case those are objects with their own thread of execution and with the goal of creating as large as possible syntactic components in which they play the head role. This would lead to a highly parallelizable parsing algorithm with many relatively independent threads. On a machine with a large number of processors, this might be an efficient approach, however in our environment the thread management overhead prevails over the effect of parallelization, so a serialized version is implemented. The implemented algorithm can easily be turned into a hybrid version as well.

In general, island parsing is more robust than the left-to-right parsing. Even when a sentence is not fully parsed, the island approach leaves more valid partial results in the chart. Island parsing can be more finely parallelized than the left-to-right parsing. In particular, there are two main advantages of island parsing when applied to HPSGs:

- efficiency reasons
  The head is typically more restrictive than other components in an HPSG rule. For this reason, it is likely that more options will be pruned right at the beginning if we start unifying daughters from the head.
- natural extension of the HPSG concept
  Starting to parse from the head follows the intuition behind the HPSG formalism. This makes the code easier to maintain, and the parser is more likely to behave as the grammar designer would expect it.

4.3. Parsing Algorithm

Two charts are used in the algorithm: a *passive chart,* and an *active chart.* The passive chart contains the completed components, while the active chart contains the equivalents of dotted rules—uncompleted components called *active chart entries.* Unlike the typical dotted rules (like in the Early's algorithm), an active chart entry contains two dots denoting the left and the right bound of the island parsed and instantiated so far. An active chart entry can expand to the left or to the right by instantiating one more daughter and moving the dot in the respective direction. When dots reach the ends of the rule, the active chart entry is completed and it is moved to the passive chart.

The parsing starts by tagging, i.e., creating one or more passive chart entries for each lexeme. Whenever a passive chart entry E is added to the passive chart, the following procedure is additionally executed:

1. **For each** rule R **do**
2.   **If** E can be unified with head of R **then**
3.   Create an active entry F
4.   **If** F is already complete **then**
5.   remove F from active chart and put it to passive chart

The main parsing algorithm is:

1. **For each** lexeme **do**
2.   add one or more passive entries

3. **While** any entries are added **do**
4.   **For each** active entry E **do**
5.   Attempt to unify one more daughter in E (i.e. move the left
      or the right dot) creating a new active entry F
6.   **If** successful **then**
7.   **If** F is complete **then** add it to passive chart
8.   **else** add it to active chart

The above description of the algorithm is somewhat simplified. In the practical algorithm, the care is taken that duplicate entries are not created in step 5. This is done by maintaining a hashtable which records all active entry-daughter pairs that are already tested.

Let us use a parsing example: Figure 2 illustrates parsing of the sentence: "What companies merged last week?" The labels on the chart entries are the head categories. The active entries are arrow-like shaped on their active side, i.e., uncompleted side. Although not shown in our example, it is possible to have "active sides" on both sides of an active entry.

5. CONCLUDING REMARKS

An HPSG Java parser is successfully implemented and described in the paper. Various issues are discussed, including a mechanism for Java-based unification and a Java island chart parser for HPSGs.

Future directions include evaluation tests with large grammars, and experiments with an actual search engine.

ACKNOWLEDGMENTS

I wish to thank Dr. Nick Cercone for valuable discussions and comments regarding this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

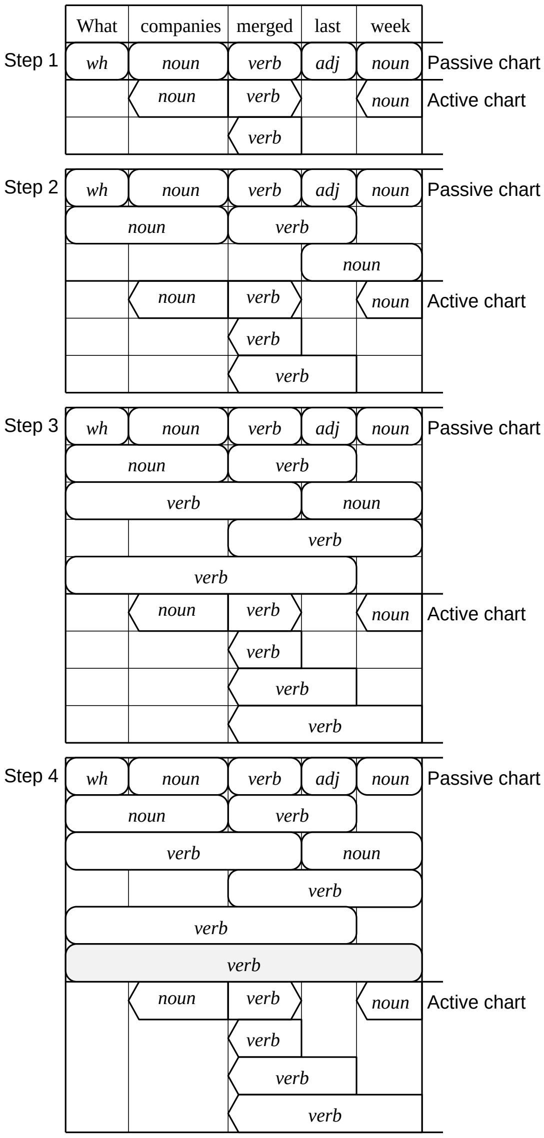

FIGURE 2. Parsing example

REFERENCES

Bob Carpenter. ALE user's guide. Laboratory for Computational Linguistics Report CMU-LCL-92-1, Laboratory for Computational Linguistics, Carnegie Mellon University, 1992.

Bob Carpenter. Probabilistic word graph parser: Java source & documentation, 1999. `http://www.colloquial.com/carp/Parser/`.

Nick Cercone, Paul McFetridge, and Fred Popowich. The SystemX natural language interface: Design, implementation and evaluation. Technical Report CSS/LCCR TR 93-03, Simon Fraser University, 1993.

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

Gary Hall. SX logical form. Unpublished manuscript. Simon Fraser Univeristy, 1998.

Takaki Makino, Minory Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. LiLFeS—towards a practical HPSG parser. In *Proceedigns of the COLING-ACL 98, Montreal*, pages 807–811, 1998.

Rob Malouf and John Carroll. Efficient graph unification for parsing constraint-based grammars. Paper presented at the Computer Science Department, Duke University, 1999. `http://hpsg.stanford.edu/rob/papers/duke-slides.ps.gz`.

David Perelman-Hall. Directed acyclic graph unification. *Dr. Dobb's Journal*, pages 44–51, 94–99, April 1995.

Carl J. Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.

Fred Popowich and Carl Vogel. Chart parsing head-driven phrase structure grammar. Technical Report CSS-IS TR 90-01, Simon Fraser University, 1990.

Derek Proudian and Carl J. Pollard. Parsing head-driven phrase structure grammar. In *Proceedings of the Twenty-Third Annual Meeting of the ACL*, pages 167–171, Chicago, IL, 1985. ACL.

Ivan A. Sag and Thomas Wasow. Syntactic theory: A formal introduction. Book draft. Stanford University, 1998.