Efficient LTAG parsing using HPSG parsers

Naoki Yoshinaga[†] **Yusuke Miyao**[†] **Kentaro Torisawa**[‡] **Jun'ichi Tsujii**[*]

[†] Dept. of Information Science, Graduate School of Science, Univ. of Tokyo

[‡] School of Information Science, Japan Advanced Institute of Science and Technology

[‡] Information and Human Behavior, PRESTO, Japan Science and Technology Corporation

[*] Dept. of Computer Science, Graduate School of Information Science and Technology, Univ. of Tokyo

[*] CREST, JST (Japan Science and Technology Corporation)

Postal address:	Dept. of Information Science, Graduate School of Science, Univ. of Tokyo, Hongo 7-3-1, Tokyo 113-0033, Japan
Telephone:	+81 3 5803 1697 Facsimile: +81 3 5802 8872
Email:	{yoshinag, yusuke, tsujii}@is.s.u-tokyo.ac.jp, torisawa@jaist.ac.jp

Summary

This paper describes a new approach for LTAG parsing using HPSG parsers. By applying a grammar conversion algorithm, we first obtain an HPSG-style grammar strongly equivalent to an original LTAG one prior to parsing. We then run an HPSG parser with the obtained HPSG-style gramamr instead of the original LTAG one. With the strongly equivalent grammars, we can make a comparison of parsing speed in both grammar formalisms. In this paper, we apply the conversion algorithm to the XTAG English grammar, which is a large-scale FB-LTAG grammar, and investigate the parsing performance in the both formalisms. Experimental results show that an efficient HPSG parser achieved a parsing speed that was higher by a factor of 25.5 than an existing LTAG parser. We show that this difference is due to the algorithmic difference in the factoring scheme in parsing.

keywords: HPSG, LTAG, parsing optimization, factoring

1 Introduction

This paper describes a parsing method for Feature-based Lexicalied Tree Adjoining Grammars (FB-LTAGs[1]) [1, 2] using parsers developed for Head-Driven Phrase Structure Grammar (HPSG) [3]. In the LTAG framework, large scale grammars have been developed for English and French [4, 5]. However, existing LTAG parsers [6, 7, 8] are practically inefficient with the large scale grammars. On the other hand in the HPSG framework, recent advantages in parsing techniques allow us to use HPSG-based processing in practical application contexts [9]. We consider that it will be beneficial that we can apply HPSG parsing techniques to LTAG parsing. As had been expected, an HPSG parser achieved a drastic speed-up against an LTAG parser in our experiment.

Since the architecture of HPSG parsing is different from that of LTAG parsing, we cannot directly apply HPSG parsers to an LTAG grammar. Therefore, we apply a grammar conversion algorithm from FB-LTAG into HPSG-style [10], and obtain an HPSG-style grammar from an LTAG grammar prior to parsing. We can directly run an HPSG parser with the obtained HPSG-style grammar instead of the original LTAG one. Since the conversion algorithm guarantees the strong equivalence, we can obtain parse results of the original LTAG grammar from those of the obtained HPSG-style grammar.

[1]In this paper, we use the term LTAG to refer to FB-LTAG, if not confusing.

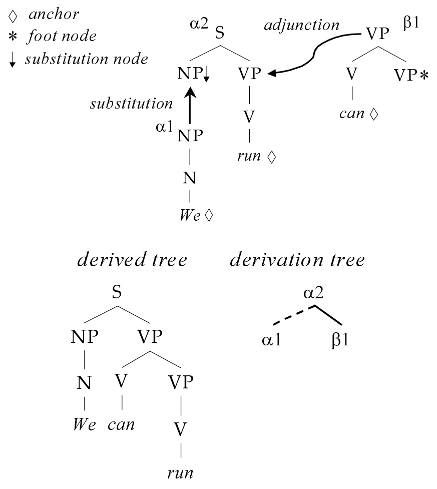


Figure 1: Tree Adjoining Grammars: basic structures and the composing operations

Parsing using the obtained HPSG-style grammars gives us an alternative parsing method for LTAGs using HPSG parsers.

In this paper, we apply the conversion algorithm to the XTAG English grammar [11], which is a large-scale FB-LTAG grammar, and investigate the parsing performance using HPSG and LTAG parsers. A result of experiments shows that an efficient HPSG parser [12] achieved the parsing speed that was higher by a factor of 25.5 than an existing LTAG parser [13]. Note that the strongly equivalent grammars enable a valid comparison of parsing speed in the both formalisms.

The efficiency of HPSG parsers compared to LTAG parsers is not evident because the theoretical bound of worst time complexity for HPSG parsing is exponential while LTAG parsing requires $O(n^6)$ for an input of length n. However, Sarkar et al. reported that the theoretical bound of computational complexity is not so significant in parsing performance for real-world texts [14]. The most dominant factor is the ambiguity of partial parse trees which is mostly caused by syntactic lexical ambiguity. We therefore expect that the LTAG and HPSG parsing techniques differ in the way they handle the syntactic lexical ambiguity.

By analyzing the experimental results, we conclude that suppressing the ambiguity of partial parse trees by the operation called *factoring* is the key to achieving high-parsing performance, and that existing HPSG parsers perform more effective factoring than the ones in LTAG parsers. Note that we have not added any special mechanisms to the parsers so that partial parse trees can be effectively factored out in the experiments. The effect caused by the difference in the factoring scheme is empirically justified by the parsing experiments using LTAG and HPSG parsers with strongly equivalent grammars.

2 Background

2.1 Feature-Based Lexicalized Tree Adjoining Grammar (FB-LTAG)

LTAG [15], an input of our algorithm, is a grammar formalism that provides syntactic analyses for a sentence by composing *elementary trees* with two operations called *substitution* and *adjunction*. Elementary trees are classified into two types, *initial trees* ($\alpha1$ and $\alpha2$ in Figure 1) and *auxiliary trees* ($\beta1$ in Figure 1). An elementary tree has at least one leaf node labeled with a terminal symbol called an *anchor* (marked with $\diamond$). In an auxiliary tree, one leaf node is labeled with the same symbol as the root node and is specially marked as a *foot node* (marked with $*$). In an elementary tree, leaf nodes with the exception of anchors and a foot node are called *substitution nodes* (marked with $\downarrow$). Substitution replaces a leaf node (*substitution node*) with another elementary tree and adjunction grafts an elementary tree with the root node and leaf node (*foot node*) labeled x onto a node of another tree with the same symbol x. FB-LTAG [1, 2] is an extension of the LTAG formalism. In FB-LTAG, each node in the elementary trees has a feature structure containing grammatical constraints on the node.

Results of analysis are described not only by *derived trees* (i.e., parse trees) but also by *derivation trees* (the right-hand side of Figure 1). A derivation tree is a structural description in LTAG and represents the history of combinations of trees.

The LTAG parser used in our experiment is a head-corner parser for general LTAGs [13]. Its

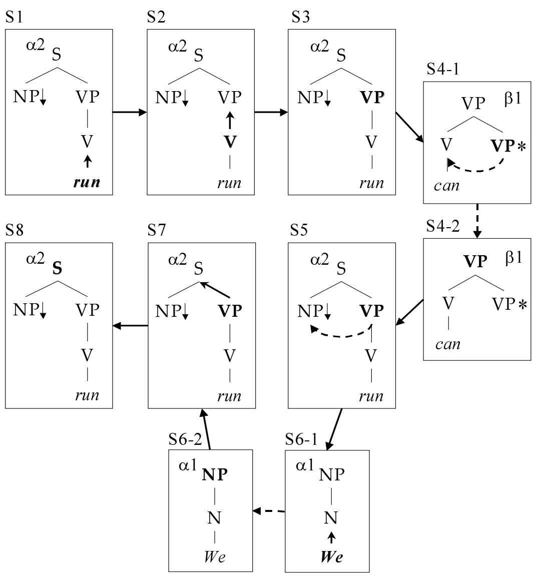


Figure 2: An example of head-corner parsing with an LTAG grammar

parsing algorithm is a chart-based variant of van Noord's [6]. The head-corner parser uses a data structure called an *agenda*. An agenda is a heap area that stores *states* to be processed. A state is essentially represented by a triplet consisting of a tree, a root node, and a processing node, which denote processed and unprocessed parts of the tree[2].

Figure 2 depicts the process of head-corner parsing for a sentence "*we can run*". In head-corner parsing, the parser traverses a tree from one leaf node called a *head-corner* to a root node. During the tree traversal, the parser recognizes the siblings of a processing node and possible adjunction at the processing node. In Figure 2, the parser first predicts an initial tree $\alpha 2$ whose root node matches the symbol S corresponding to a sentence (the state s1 in Figure 2). The parser proceeds by moving in a bottom-up manner from the head-corner "*run*" to S. Next, after moving up to the node VP in $\alpha 2$ (the state s3 in Figure 2), the parser recognizes adjunction at the processing node VP, and then introduces a new state s4-1 for the adjoining tree $\beta 1$. After recognizing $\beta 1$ (the state s4-2 in Figure 2), the parser tries to recognize the sibling node of VP (the state s5 in Figure 2). To recognize the sibling node NP, the parser introduces a new state s6-1 for $\alpha 1$. Then, the parser proceeds to the root node S of $\alpha 2$ (the state s8 in Figure 2). Since there is no state to be processed in the agenda, parsing for a sentence "*we can run*" completes.

[2]States include other parameters such as spans over an input string, though we do not give a detail.

The head-corner parser performs factoring when the parser generates a new state. The parser checks whether a state equivalent to the new state already exists in the agenda or not. The parser pushes the new state in the agenda only when the equivalent state does not exist in the agenda. This enables to avoid generating duplicated equivalent partial parse trees.

2.2 Head-Driven Phrase Structure Grammar (HPSG)

We define an HPSG-style grammar, which is an output of the conversion, as a unification-based grammar to which existing HPSG parsers can apply. An HPSG-style grammar consists of *lexical entries* and *ID grammar rules*, each of which is described with typed feature structures [16]. A lexical entry for each word expresses the characteristics of the word, such as the subcategorization frame and the grammatical category. An ID grammar rule represents a relation between a mother and its daughters, and is independent of lexical characteristics.

The HPSG parsers used in the experiment are CKY-based HPSG parsers [17]. The CKY-based HPSG parser uses a data structure called a *triangular table*. A triangular table is a heap area that stores *edges*, which correspond to partial parse trees. An edge is described with a feature structure, and is stored in a cell in the triangular table.

Figure 3 illustrates an example of CKY-based parsing with an HPSG grammar. First, lexical entries for "*we*", "*can*", and "*run*" are stored as edges e1, e2, and e3 in the triangular table. Next, edges e2 and e3 are unified respectively with the daughter feature structures of an ID grammar rule. The feature structure of the mother node is determined as a result of these unifications, and is stored as a new edge e4. Then an ID grammar rule is applied to edges e1 and e4, and an edge e5 is generated. Since the parse tree spans the

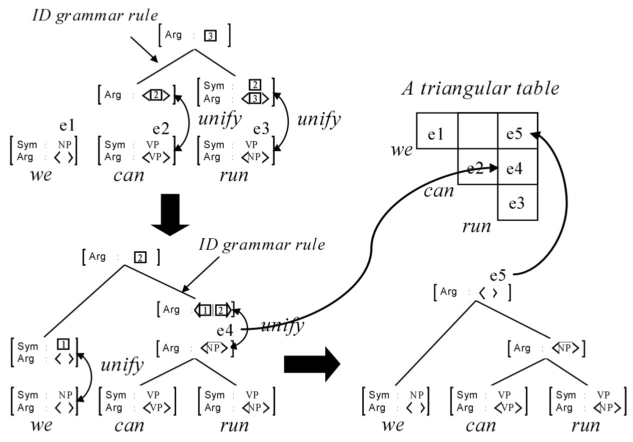
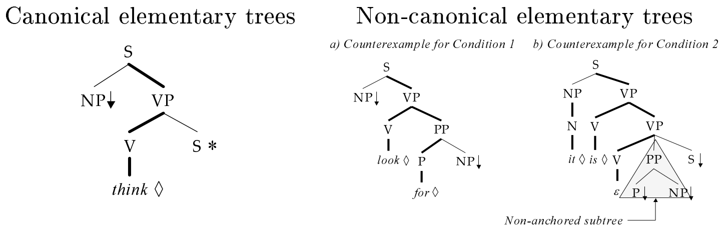


Figure 3: An example of CKY-based parsing with an HPSG grammar



Canonical elementary trees
Non-canonical elementary trees
a) Counterexample for Condition 1
b) Counterexample for Condition 2
think ◊
look ◊
for ◊
it ◊ is ◊
Non-anchored subtree

Figure 4: A canonical elementary tree and exceptions

whole input string, parsing for a sentence "*we can run*" completes.

The CKY-based parser performs factoring when the parser generates a new edge. The parser checks whether an edge equivalent to the new edge already exists in the cell in the triangular table, and store the equivalent edge in the cell only when the equivalent edge does not exist in the triangular table. This enables to avoid generating duplicated equivalent partial parse trees as in LTAG parsing.

3 Grammar conversion

In the following, we introduce the grammar conversion algorithm [10]. Given the definitions in Section 2, the algorithm converts LTAG grammars into HPSG-style grammars by converting elementary trees into HPSG lexical entries, and emulates substitution and adjunction by predetermined ID grammar rules.

The algorithm converts LTAG elementary trees into HPSG lexical entries 1) by converting

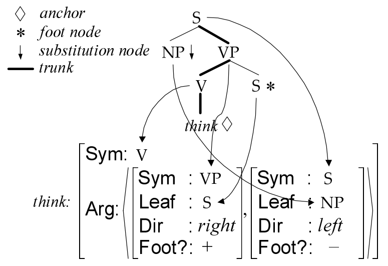


Figure 5: A conversion from a canonical elementary tree into an HPSG lexical entry

all LTAG elementary trees into tree structures called *canonical elementary trees* (the left-hand side of Figure 4) which correspond to HPSG lexical entries one-to-one, and 2) by converting a canonical elementary tree into an HPSG lexical entry. Canonical elementary trees must satisfy the conditions that a tree must have only one anchor (Condition 1), and that all branchings in a tree must contain *trunk nodes* (trunk nodes are internal nodes on a *trunk*, which is a path from an anchor to the root node.) (Condition

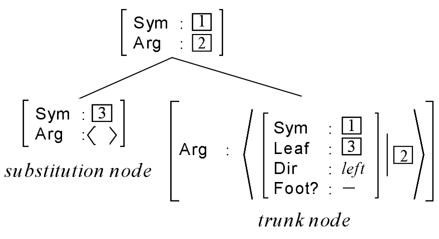


Figure 6: Left substitution rule

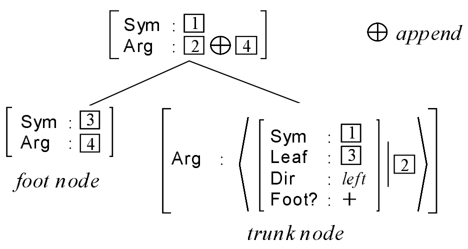


Figure 7: Left adjunction rule

2). Condition 1 guarantees that a canonical elementary tree has only one trunk, and Condition 2 guarantees that each branching consists of a trunk node, a leaf node, and their mother (also a trunk node).

Figure 5 depicts a conversion of *canonical elementary trees*. A canonical elementary tree is converted to an HPSG lexical entry by storing each leaf node as subcategorization elements in a stack (the Arg feature in Figure 5) where each leaf node is expressed by a triplet consisting of the symbol, the direction against the trunk, and the type (the Leaf, Dir, and Foot? features in Figure 5, respectively). Together with the leaf node, the symbol of a trunk node is stored as the Sym feature in order to explicitly determine the symbol of a mother node in rule applications.

Substitution and adjunction are emulated by grammar rules described in Figure 6 and 7, respectively[3]. A substitution rule pops an element in the value of the Arg feature and lets the trunk node subcategorize a node unifiable with the popped element. An adjunction rule concatenates the values of the Arg features of both daughters.

Figure 8 shows an instance of rule applica-

[3]A co-indexing box ($\boxed{n}$) expresses that the two subfeature structures are sharing their values with each other.

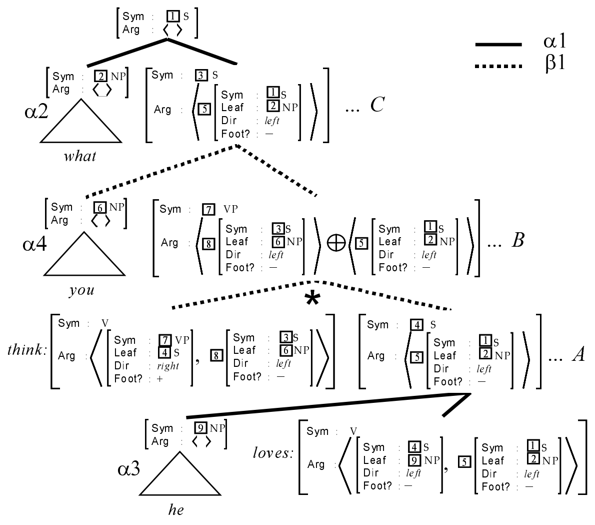


Figure 8: An example of rule applications

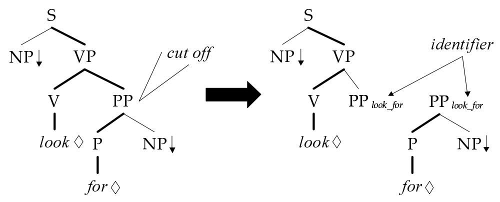
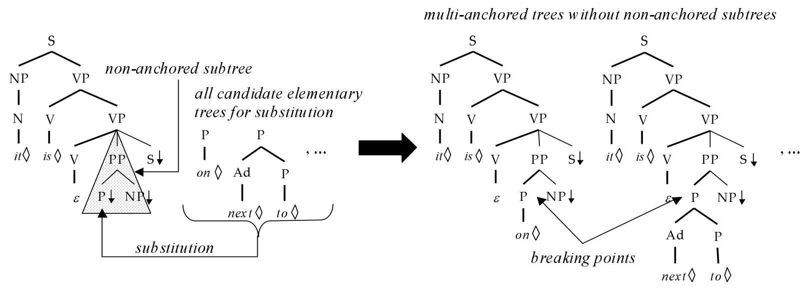


Figure 9: Division of a multi-anchored elementary tree into single-anchored trees

tions. The thick line indicates the adjoined tree ($\alpha1$) and the dashed line indicates the adjoining tree ($\beta1$). The adjunction rule is applied to construct the branching marked with ⋆, where "*think*" takes as an argument a node whose Sym feature's value is S. By applying the adjunction rule, the Arg feature of the mother node (B) becomes a concatenation list of both Arg features of $\beta1$ ($\boxed{8}$) and $\alpha1$ ($\boxed{5}$). Note that when the construction of $\beta1$ is completed, the Arg feature of the trunk node (C) will be its former state (A). We can continue constructing $\alpha1$ as if nothing had happened.

Non-canonical elementary trees (the right-hand side of Figure 4) are converted into canonical ones, to which the above algorithm can be applied. Multi-anchored elementary trees, which violate Condition 1, are divided into multiple canonical elementary trees at an internal node (*cut-off node*) (Figure 9). Note that a cut-off node is marked by an *identifier* to preserve a co-



non-anchored subtree
all candidate elementary trees for substitution
substitution
multi-anchored trees without non-anchored subtrees
breaking points

Figure 10: Conversion of a tree with non-anchored subtrees into multi-anchored trees without non-anchored subtrees

occurrence relation among the multiple anchors.

Non-canonical elementary trees violating Condition 2 have a *non-anchored subtree* which is a subtree of depth 1 or above with no anchor. A non-anchored subtree is converted into multi-anchored trees by substituting the deepest node in the non-anchored subtree with elementary trees (Figure 10). Substituted nodes are marked as *breaking points* to remember that the nodes originate from the substitution nodes. In the resulting trees, all subtrees are anchored so that we can apply the above conversion algorithms.

The above algorithm gives the conversion of an LTAG grammar, and can be easily extended to handle an FB-LTAG grammar by merely storing a feature structure of each node into the Sym feature and Leaf feature together with the non-terminal symbol. Feature structure unification is executed by ID grammar rules.

The strong equivalence is assured because only substitution/adjunction operations performed in LTAG are performed with the obtained HPSG-style grammar. This is because each element in the Arg feature selects only feature structures corresponding to trees which can substitute/adjoin each leaf node of an elementary tree. By following a history of rule applications, each combination of elementary trees in LTAG derivation trees can be readily recovered.

The strong equivalence also holds for conversion of non-canonical elementary trees. For trees violating Condition 1, we can distinguish the cut-off nodes from the substitution nodes owing to identifiers, which recover the co-occurrence relation in the original elementary trees between the divided trees. For trees violating Condition 2, we can identify substitution nodes in a combined tree because they are marked as breaking points, and we can consider the combined tree as two trees in the LTAG derivation.

Table 1: Parsing performance with the XTAG English grammar for the ATIS corpus

Parser	Parse Time (sec.)
Naive	3.32
TNT	0.77
lem	19.64

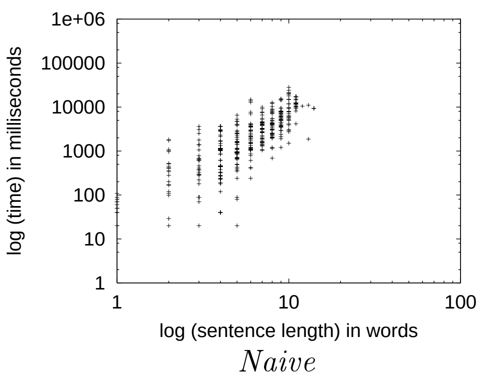
4 Experiments

We applied the above algorithm to the latest version of the XTAG English grammar [11][4], which is a large-scale LTAG grammar for English. We successfully converted all the elementary trees[5] in the XTAG English grammar to HPSG lexical entries[6]. We obtained exactly the same derivation trees by using the original and the obtained grammars in the parsing experiment with 457

[4]We used the grammar attached to the latest distribution of an LTAG parser which we used for the parsing experiment. The parser is available at: ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.13.0.i686.tgz

[5]*Elementary trees* should be in fact denoted as *elementary tree templates*. That is, elementary trees are abstracted from lexicalized trees, and one elementary tree template is defined for one syntactic construction, which is assigned to number of words.

[6]In the following experiments, we eliminated 32 elementary trees because the LTAG parser cannot produce correct derivation trees with them.



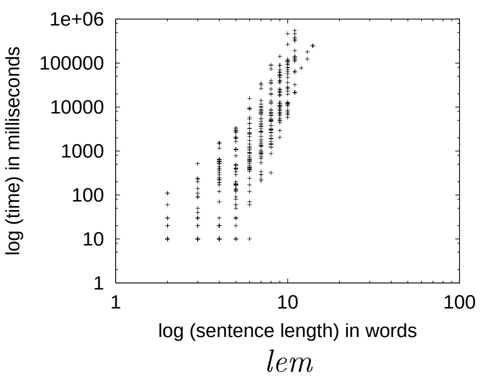


Figure 11: Parsing performance with the XTAG English grammar for the ATIS corpus

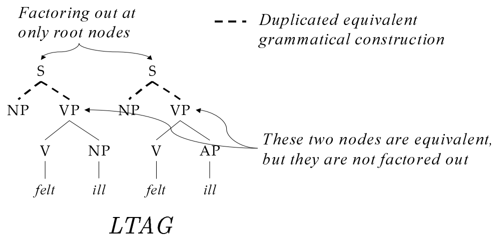
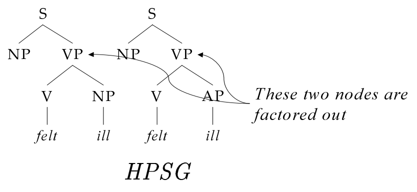

Factoring out at only root nodes
Duplicated equivalent grammatical construction
S
S
NP
VP
NP
VP
V
NP
V
AP
felt
ill
felt
ill
These two nodes are equivalent, but they are not factored out
LTAG


Figure 12: The difference of the factoring scheme between LTAG and HPSG parsing

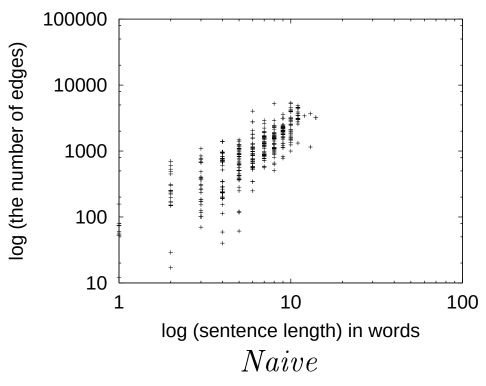
sentences from the ATIS corpus [18][7] (the average length is 6.32 words). This result empirically attests the strong equivalence of the grammar conversion.

Table 1 shows the parsing speed. In Table 1, *lem* refers to the LTAG parser [13], ANSI C implementation of the two-phase parsing algorithm that performs the head corner parsing [6] without features (phase 1), and then executes feature unification (phase 2). *Naive* refers to the CKY [19]-based HPSG parser without optimization techniques, and *TNT* refers to the HPSG parser [12], C++ implementation of the two-phase parsing algorithm that performs filtering with a compiled CFG (phase 1) and then executes feature unification (phase 2). The result shows that the HPSG parser achieved a speed-up by a factor of 25.5.

Figure 11 shows the parse time plotted against a sentence length, where both in logarithmic scales. Since the increase of parse time plotted against a sentence length n in logarithmic scales is equal to the degree of polynomial order of the empirical time complexity, the graphs show the order of the empirical time complexity of *lem* is higher than that of *Naive*. By observing the algorithmic difference of both parsers, we found the following two differences in the parsing scheme were major factors to cause the differences in the empirical time complexity.

One factor is the difference of the *factoring* scheme in both parsers. As noted in Section 2, factoring is a commonsensical parsing technique to avoid generating duplicated equivalent partial parse trees [20], and enables us to reduce the time complexity of parsing from exponential to polynomial. Both of the parsers have the architecture for performing factoring, but the ways of factoring are different. The HPSG parser treats a branching as its minimal component, and performs factoring when the mothers of a branching are equivalent to each other (the right-hand side of Figure 12). On the other hand, the LTAG parser treats an elementary tree as its minimal

[7]In the experiments, we eliminated 59 sentences because of a time-out of the parsers, and 61 sentences because the LTAG parser does not produce correct derivation trees because of bugs in its preprocessor.



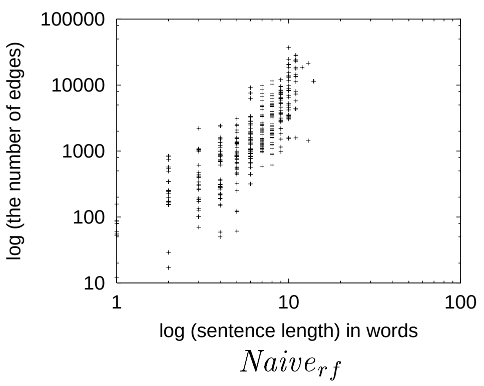


Figure 13: The number of edges of *Naive* (left) and a variant of *Naive* ($Naive_{rf}$) which performs factoring only when the factoring can be performed in *lem* (right)

component, and performs factoring only for elementary trees with equivalent root nodes (the left-hand side of Figure 12). Since the root node corresponds to a feature structure whose Arg feature is a null list in HPSG, this difference means that the HPSG parser factors out more partial parse trees than the LTAG parser. As illustrated in Figure 12, the LTAG parser cannot avoid duplicating equivalent grammatical constructions corresponding to fragments of elementary trees. This difference of the factoring scheme causes the difference of the empirical time complexity.

To verify the above argument, we made a parsing experiment with the same corpus by using a variant of *Naive* (hereafter $Naive_{rf}$) which performs factoring only when the factoring can be performed by the LTAG parser[8]. As stated in Section 2, partial parse trees are stored as a set of *edges* in the HPSG parser and *states* in the LTAG parser, respectively. Hence, the number of edges/states generated by the parsers is one of common indicators of the empirical time complexity [20]. Figure 13 shows the number of edges plotted against a sentence length, where both in logarithmic scales[9]. The increase of the number of edges of $Naive_{rf}$ was higher than that of *Naive*. Since the parsing scheme of $Naive_{rf}$ is rather similar to that of *lem*, the difference of parse time between *Naive* and *lem* and the difference of the number of edges between *Naive* and $Naive_{rf}$ tell that the difference of the factoring scheme was the major cause of difference of the empirical time complexity.

The other factor is the effective CFG filtering. *TNT* filters out most impossible partial parse trees by using a compiled CFG [21, 22][10]. While Poller and Becker suggest a CFG filtering for LTAG parsing [7], CFG filtering for HPSG parsing has an advantage over their method. Their method extracts CFG from an LTAG grammar by regarding each branching in an elementary tree as a CFG rule, that is, non-terminal symbols of CFG correspond to the nodes in an elementary tree. On the other hand, in HPSG, since the conversion encodes all branchings in an elementary tree into a feature structure, non-terminals of a compiled CFG, which correspond to HPSG lexical/phrasal signs, contain the whole

[8]Although the comparison between the states of the LTAG parser and the edges of the HPSG parsers may be more appropriate, we do not give them because the state of lem is that of its phase 1, and each edge of the HPSG parsers does not have a one-to-one correspondence with each state of the LTAG parsers. In this context, we also do not show an experiment using *TNT* because the advantage of *TNT* is not only because of the effective factoring but also because of the effective CFG filtering we mention below.

[9]We should mention that $Naive_{rf}$ in fact performs some kind of factoring which is not performed by the LTAG parser. Although this may suppress the number of edges of $Naive_{rf}$, it has no serious effect on our conclusion.

[10]We should note that we required another technique [23] to compile the obtained HPSG grammar into CFG. The technique allowed us to compile any HPSG-style grammar into the reasonable size of CFG.

structure of the elementary tree. This is supported by the fact that the number of non-terminal symbols in the compiled CFG is 7,553 against 19 in the original LTAG grammar[11]. Although no empirical result has been reported using the CFG filtering for LTAG parsing, we conclude that the CFG filtering in HPSG parsing restricts candidate parse trees more strictly than that of LTAG parsing.

5 Related work

There are two works on grammar conversion between LTAG and HPSG. Tateisi et al. also translated LTAG into HPSG [24]. However, their method depended on translator's intuitive analysis of the original grammar, and thus the translation was manual and grammar dependent. The manual translation demanded considerable efforts from the translator, and obscures the strong equivalence between the original and obtained grammars. Other works converted HPSG into LTAG [25, 26]. Given the greater generative power of HPSG, the conversion required some restrictions on HPSG to suppress its generative capacity. As a result, the conversion loses the strong equivalence of the grammars. The existing works are inappropriate for our parsing method, which requires a grammar conversion which guarantees strong equivalence.

There is another work on a comparison between HPSG and LTAG parsers. Yoshida et al. reported a comparison using HPSG and LTAG parsers on the same platform (LiLFeS [27] programming language) [8], but it was not a fair comparison because an HPSG grammar they used was a translation of a subset of the LTAG grammar [24]. As a consequence, their comparison is meaningless as an empirical comparison of HPSG and LTAG parsers. We again say that the comparison of parsers in different grammar formalisms is meaningful only when we perform the experiments using the strongly equivalent grammars.

[11]We should note that grammatical constraints given in feature structures of nodes in elementary trees are not considered when we compile the obtained HPSG-style grammar into CFG.

6 Conclusion

In this research, we described a parsing method for LTAGs using parsers developed for HPSG. We applied the conversion algorithm to the latest version of the XTAG English grammar and performed the parsing experiments using existing HPSG and LTAG parsers with strongly equivalent grammars. We observed that an existing HPSG parser achieved a drastic speed-up by a factor of 25.5 against an existing LTAG parser. By observing how HPSG and LTAG parsing techniques have an effect on the empirical parse speed, we investigated the reason for the efficiency of HPSG parsers. We concluded that suppressing the ambiguity of partial parse trees by the operation called factoring is the most important factor in achieving high-parsing performance, and that existing HPSG parsers are more beneficial than LTAG ones in this respect. Our parsing method enables us not only to achieve efficient LTAG parsing but also to observe the algorithmic difference between parsers in different grammar formalisms. We can also evaluate the empirical effect of the algorithmic difference on the parse speed.

Our study can also suggest how HPSG parsing techniques should be realized in LTAG parsing. In the case of the factoring scheme, we can optimize LTAG parsing by merging two different states which have an equivalent unprocessed part into one state. This optimization must reduce a significant number of states, which contributes to the considerable speed-up. In the case of the CFG filtering, although we did not show the algorithmic difference in detail, the same argument as the factoring scheme must be possible.

References

[1] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars.* PhD thesis, Department of Computer & Information Science, University of Pennsylvania, 1987.

[2] K. Vijay-Shanker and Aravind K. Joshi. Feature structures based Tree Adjoining Grammars. In *Proc. of 12th COLING '92*, pages 714–719, 1988.

[3] Carl Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press and CSLI Publications, 1994.

[4] Christy Doran, Beth Ann Hockey, Anoop Sarkar, B. Srinivas, and Fei Xia. Evolution of the XTAG system. In Anne Abeillé and Owen Rambow, editors, *Tree Adjoining Grammars: Formal, Computational and Linguistic Aspects*, pages 371–403. CSLI publications, 2000.

[5] Anne Abeillé and Marie-Hélène Candito. FTAG: A Lexicalized Tree Adjoining Grammar for French. In Anne Abeillé and Owen Rambow, editors, *Tree Adjoining Grammars: Formal, Computational and Linguistic Aspects*, pages 305–329. CSLI publications, 2000.

[6] Gertjan van Noord. Head corner parsing for TAG. *Computational Intelligence*, 10(4):525–534, 1994.

[7] Peter Poller and Tilman Becker. Two-step TAG parsing revisited. In *Proc. of TAG+4*, pages 143–146, 1998.

[8] Minoru Yoshida, Takashi Ninomiya, Kentaro Torisawa, Takaki Makino, and Jun'ichi Tsujii. Efficient FB-LTAG parser and its parallelization. In *Proc. of PACLING '99*, pages 90–103, 1999.

[9] Dan Flickinger, Stephen Oepen, Jun'ichi Tsujii, and Hans Uszkoreit, editors. *Natural Language Engineering – Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation.* Cambridge University Press, 2000.

[10] Naoki Yoshinaga and Yusuke Miyao. Grammar conversion from FB-LTAG to HPSG. In *Proc. of the Sixth ESSLLI Student Session*, 2001. To appear.

[11] The XTAG Research Group. A Lexicalized Tree Adjoining Grammar for English. http://www.cis.upenn.edu/~xtag/, 2001.

[12] Kentaro Torisawa, Kenji Nishida, Yusuke Miyao, and Jun'ichi Tsujii. An HPSG parser with CFG filtering. *Natural Language Engineering – Special Issue on Efficient Processing with HPSG: Methods, Systems, Evaluation*, 6(1):63–80, 2000.

[13] Anoop Sarkar. Practical experiments in parsing using Tree Adjoining Grammars. In *Proc. of TAG+5*, pages 193–198, 2000.

[14] Anoop Sarkar, Fei Xia, and Aravind Joshi. Some experiments on indicators of parsing complexity for lexicalized grammars. In *Proc. of COLING 2000*, pages 37–42, 2000.

[15] Yves Schabes, Anne Abeille, and Aravind K. Joshi. Parsing strategies with 'lexicalized' grammars: Application to Tree Adjoining Grammars. In *Proc. of 12th COLING '92*, pages 578–583, 1988.

[16] Bob Carpenter. *The Logic of Typed Feature Structures.* Cambridge University Press, 1992.

[17] Andrew R. Haas. Parallel parsing for unification grammars. In *Proc. of IJCAI '87*, pages 615–618, 1987.

[18] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1994.

[19] Tadao Kasami. An efficient recognition and syntax algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab., Bedford, Mass., 1965.

[20] Martin Kay. Algorithm schemata and data structures in syntactic processing. Technical Report CSL-90-12, Xerox Palo Alto Research Center, Palo Alto, CA, 1980.

[21] Kentaro Torisawa and Jun'ichi Tsujii. Computing phrasal-signs in HPSG prior to parsing. In *Proc. of COLING '96*, pages 949–955, 1996.

[22] Bernd Kiefer and Hans-Ulrich Krieger. A Context-Free approximation of Head-Driven Phrase Structure Grammar. In *Proc. of IWPT 2000*, pages 135–146, 2000.

[23] Kentaro Torisawa, Yusuke Miyao, and Naoki Yoshinaga. Yet another technique for CFG filtering in HPSG parsing, 2001. In preperation.

[24] Yuka Tateisi, Kentaro Torisawa, Yusuke Miyao, and Jun'ichi Tsujii. Translating the XTAG English grammar to HPSG. In *Proc. of TAG+4*, pages 172–175, 1998.

[25] Robert Kasper, Bernd Kiefer, Klaus Netter, and K. Vijay-Shanker. Compilation of HPSG to TAG. In *Proc. of ACL '95*, pages 92–99, 1995.

[26] Tilman Becker and Patrice Lopez. Adapting HPSG-to-TAG compilation to wide-coverage grammars. In *Proc. of TAG+5*, pages 47–54, 2000.

[27] Takaki Makino, Minoru Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. LiLFeS — towards a practical HPSG parsers. In *Proc. of COLING–ACL '98*, pages 807–811, 1998.