Compiling an HPSG-based grammar into more than one CFG

Kenji Nishida
Department of Information Science, Graduate School of Science, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan, 113-0033
(Tel) +81-3-5803-1697, (Fax) +81-3-5802-8872
nishiken@is.s.u-tokyo.ac.jp

Kentaro Torisawa
Graduate School of Information Science, Japan Advanced Institute of Science and Technology
Asahidai 1-1, Tatsunokuchi-cho, Noumi-gun, Ishikawa, 923-1292, Japan
Information and Human Behavior, PRESTO, Japan Science and Technology Corporation
Kawaguchi Hon-cho 4-1-8, Kawaguchi-shi, Saitama, 332-0012, Japan
torisawa@jaist.ac.jp

Jun'ichi Tsujii
Department of Information Science, Graduate School of Science, University of Tokyo
Hongo 7-3-1, Bunkyo-ku, Tokyo, Japan, 113-0033
(Tel) +81-3-5803-1697, (Fax) +81-3-5802-8872
CREST,JST(Japan Science and Technology Corporation)
Kawaguchi Hon-cho 4-1-8, Kawaguchi-shi, Saitama, 332-0012, Japan
tsujii@is.s.u-tokyo.ac.jp

Abstract

Recently, the performance of HPSG parsing has been improved so that the parsers can be applied to real-world texts. CFG filtering is one of the techniques which contributed to this progress. It improved parsing speed by filtering impossible parse trees by using the CFG compiled from a given HPSG-based grammar. However, there is a limit in the speed-up. This is because the compiled CFG grows into an impractical size when the original grammar is complex. This caused serious memory problems in actual parsing. This paper describes a technique to avoid this obstacle. We compile a given HPSG-based grammar to more than one CFG, and utilize all the CFGs in a filtering process. We show that, by this technique, we can achieve almost the same parsing speed with far less memory through a series of experiments.

Keywords: parsing algorithm, HPSG, grammar compilation

1 Introduction

Recent advances in HPSG parsing [1] have enabled us to apply it to real-world texts, and they opened the way to obtain a deep syntactic analysis of real-world texts with the high-level grammar formalism [2]. One of the techniques that contributed to this progress is the method called *CFG filtering* [3] [4]. It improved parsing speed by filtering impossible parse trees by using the CFG compiled from a given HPSG-based grammar. However, there is a limit in the speed-up. The compiled CFG grows into an impractical size

as the original grammar becomes complex. This causes a serious memory problem in actual parsing. Although we can handle this situation by sacrificing the precision of filtering in principle, it implies slow down of a parsing process and prevents us from achieving further speed-up.

We have already proposed a technique called *Array Unification* [5], which is a simplified unification that can augment CFG filtering. Certainly, the cost of array unification is smaller than that of full unification, and we could achieve significant speed-up. However, the filtering procedure with array unification is still expensive compared to CFG filtering, and in order to achieve higher speed-up, we need to improve CFG filtering itself.

In this study, we compile an HPSG-based grammar into more than one distinct CFG. Those CFGs replace a single CFG in the original CFG filtering. We show that, by this technique, we can achieve almost the same parsing speed with less memory through a series of experiments.

The next section overviews the original CFG filtering. Our algorithm is described in Section 3. Section 4 shows experimental results. And Section 5 gives concluding remarks and future works.

2 Overview of Original CFG Filtering

This section overviews the original CFG filtering. We assume that an HPSG-based grammar consists of two components, namely, rule schemata and lexical entries. Both are given in the form of feature structures. Rule schemata correspond to rewriting rules in a CFG, and its application is done by unification of feature structures.

Compilation of an HPSG-based grammar to a CFG is summarized as the following steps.

Step 1 Apply the rule schemata to a lexical entry assuming that it is a head daughter, and construct partial parse trees (Figure 1). Note that equivalent partial parse trees are factored out into a single partial parse tree in this process. [1]

Step 2 Derive a CFG from partial parse trees (See Figure 1).

(a) Assign nonterminals of a CFG to feature structures in a partial parse tree.

(b) Check whether a partial parse tree can connect to another partial parse tree by unification.

(c) Generate a rewriting rule such that it can generate the connection of partial parse trees.

A parsing process consists of two phases: Phase 1 and Phase 2. Phase 1 is a CFG parsing with a compiled CFG. It constructs candidates for parse trees by CFG parsing. At Phase 2, the parser applies the original rule schemata to the completed parse tree candidates generated by the CFG by means of unification. Note that the parser does not have to attempt all the possible unification. Considerable portion of impossible parse trees are eliminated by the compiled CFG during Phase 1. The parser does not have to waste the time that would be wasted in the unification to be failed. This leads to speed-up of a whole parsing process. In the most successful case, Torisawa et al. reported about 40 times speed-up. [3]

The problem is that the above compilation algorithm may not terminate, or, even if it terminates, it may produce a huge CFG such that it cannot be stored on actual machines. This is because some grammars can generate enormous number of partial parse trees in the compilation process. The set of all the partial trees may have infinite cardinality. In order to avoid this problem, Torisawa et al. applied the technique called *Shieber's restriction* [6], which eliminates the values of certain features. This enables us to avoid the termination problem and to control the size of a CFG. However, if we apply the restriction, the precision of filtering must be sacrificed, yielding the parsing speed slow down. This was an obstacle to achievement of further speed-up. The next section gives an extended algorithm which can avoid this obstacle.

[1] Actually, partial parse trees generated at this step are represented in a graph structure. Equivalent feature structures are factored out into a single feature structure in this representation. Then, some of infinitely tall partial parse trees can be *enumerated* in a finite amount of time.

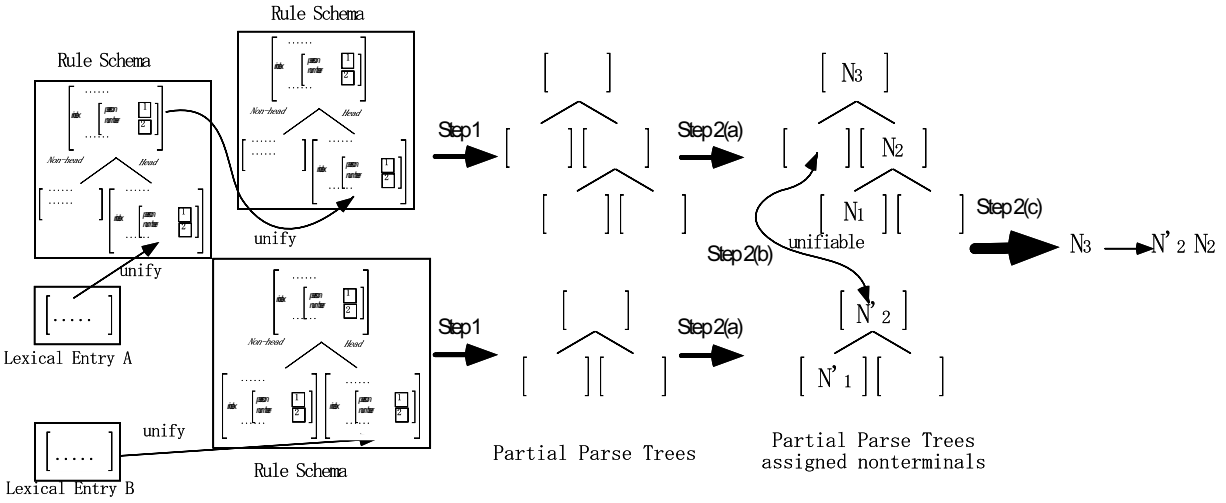

Figure 1: Derivation of a CFG rule

3 Multiple CFG Filtering

This section describes our method which can avoid the difficulty described in the previous section. We call this method *multiple CFG filtering*. The main idea is to utilize more than one CFG in filtering. Each CFG is obtained by applying different restriction scheme to original HPSG-based grammar, i.e., by eliminating different sets of features from the grammar.

Assume that an HPSG-based grammar in Figure 2 is given. Each lexical entry contains two features *person* and *number*. Consider the situation that the features can have following values, and that one of six combinations of those values appears in a lexical entry.

- *person* : 1st, 2nd, 3rd
- *number* : single, plural

Figure 3 illustrates the compilation processes of the grammar with different restrictions. (A) shows compilation without any restriction, i.e, no features are eliminated from the grammar, (B) and (C) give the compilation with some restriction. (B) is the compilation with eliminating the *person* feature, and (C) is the one with removing the *number*. Since equivalent partial parse trees are factored out during compilation processes, the resulting partial parse trees of (A) are larger than those of (B) and (C). Note that, in the compilation to CFG, the feature structures in the generated partial parse trees have one-to-one correspondences with a resulting nonterminal in the CFG. This means that the CFGs generated in (B) and (C), namely G_B and G_C respectively, have smaller numbers of nonterminals than G_A obtained in (A).

The main idea here is to replace the large single grammar G_A with the two smaller grammars, G_B and G_C. Note that the check of the *person* values is not performed in G_B, but it can be done by G_C. As for the feature *number*, the opposite holds. This means that G_B and G_C together represent the same constraints as that expressed by G_A.[2] If we can obtain reasonably small grammars G_B and G_C, we can achieve almost the same filtering performance as G_A with far less memory. This observation is confirmed by the experiments given in the next section. Note that we can generalize this observation to the cases that we replace a single CFG with more than *two* CFGs. In other words, the improvement of memory efficiency can be achieved by replacing a single CFG with more than two CFGs.

Now, we can proceed to the parsing scheme with multiple CFG filtering. We adopted CKY-based parsing [7] as a parsing algorithm for CFG. This parsing scheme uses a device called a CKY

[2] It generally stands up except the case that there are structure sharings between the *number* feature and the *person* one.

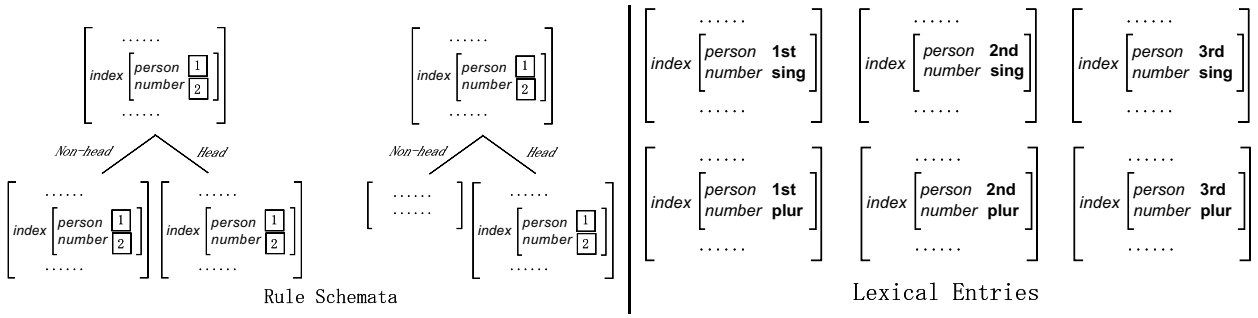

Figure 2: A sample HPSG grammar

table, which is a triangular table consisting of the cells corresponding to substrings in a input sentence, and it repeatedly generates *edges* in each cell. Each cell is indexed by the substring it covers. More precisely, a cell is denoted by $C_{i,j}$ where i and j are the positions in an input string. $C_{i,j}$ contains the edges covering the substrings spanning from the i-th word to the $j-1$-th word in the input sentence. Each edge is denoted by a tuple $\langle id, NT, dtr_1, dtr_2 \rangle$, where id is an identifier of the edge, NT is a nonterminal symbol, dtr_1 and dtr_2 are identifiers of *daughter* edges.(We assume that all the rewriting rules in a given CFG is binary-branching. Then, all edges have to keep *two* daughters.)

The parsing is done by applying rewriting rules to every possible combination of two edges. Given two edges in a CKY-table, $\langle id, NT, dtr_1, dtr_2 \rangle$ in the cell $C_{i,j}$ and $\langle id', NT', dtr'_1, dtr'_2 \rangle$ in $C_{j,k}$. If there is a rewriting rule in the form of $M \rightarrow NTNT'$ in a given CFG, the parser generates the edge $\langle id_{new}, M, id, id' \rangle$ and stores it into the cell $C_{i,k}$. This process is repeated until no more edges can be generated.

We need to extend the above CKY-based parsing scheme so that it can handle more than one CFG simultaneously. Assume that we are given N compiled CFGs denoted by $G_1, G_2, \cdots, G_N$ at Phase 1. We use edges with N nonterminal symbols. An edge is denoted by a tuple $\langle id, NT_1, NT_2, \cdots, NT_N, dtr_1, dtr_2 \rangle$ where NT_i is a nonterminal symbol in the grammar G_i for any integer i $(1 \leq i \leq n)$. Intuitively, what the extended CKY parsing does is to generate an edge which is *licenced* by all the given grammars.

Assume that there exist two edges, namely $\langle id, NT_1, NT_2, \cdots, NT_N, dtr_1, dtr_2 \rangle$ in the cell $C_{i,j}$ and $\langle id', NT'_1, NT'_2, \cdots, NT'_N, dtr'_1, dtr'_2 \rangle$ in $C_{j,k}$. The new parsing scheme generates a new edge $\langle id_{new}, M_1, M_2, \cdots, M_N, id, id' \rangle$ if and only if the following condition is satisfied.

- For every integer i $(1 \leq i \leq n)$, there exists a rewriting rule $M_i \rightarrow NT_i NT'_i$ in G_i.

In actual implementation, if we use the above representation of the edge, the number of edges in parsing can be quite large. We need to introduce the mechanism which factors out the edges sharing the same nonterminal symbols. This is done by using more than one CKY table, each of which is used for a single CFG, and by linking the corresponding edges in distinct CKY tables so that the linked edges can express the same information as that of a edge in the above format. This contributed to a significant speed-up. Figure 4 shows an example of factoring.

Now, we have a mechanism to treat multiple CFGs compiled from a single HPSG-based grammar. The next section shows that this mechanism works effectively and achieves almost the same speed-up as that of the original CFG filtering.

4 Experiments

We compiled a wide-coverage HPSG-based grammar, XHPSG [8], into three different CFGs: CFG (a), CFG (b), and CFG (c). CFG (a) and CFG (b) have been obtained by eliminating more features than in the compilation of CFG (c). In

	a	b	c
Number of nonterminals	1682	4821	7694
Number of rewriting rules	47516	1537495	2035616
Eliminated features	*ASSIGN_CASE* *ASSIGN_COMP* *ASSIGN_XP* *CASE* *CFORM* *CONT* *EXTRACT* *GERUND* *HEADPHON* *INV* *MAINV* *MARKING* *PASS* *PREF* *PFROM* *PHONOLOGY* *PRD* *PROG* *PRON* *P_ASSIGN_CASE* *REFL* *REL_CLAUSE* *TENSE* *TRANS* *TRF* *VMODE* *XP*	*ASSIGN_COMP* *ASP* *CONT* *HEADPHON* *INHER* *MAINV* *MARKING* *PHONOLOGY* *REFL* *SPR* *TENSE* *TRF* *WH*	*ASSIGN_COMP* *CONT* *HEADPHON* *MAINV* *MARKING* *PHONOLOGY* *REFL* *TENSE* *TRF*

Table 1: CFGs used in the experiments

CFG	Phase 1(msec)	Phase 2(msec)	Phase 1 + Phase 2(msec)	used memory(MB)
(a)	46	770	816	175
(b)	81	520	601	431
(a)+(b)	64	110	174	481
(c)	62	109	171	845

Table 2: Comparison of average parsing time(msec) per a sentence and memory usage

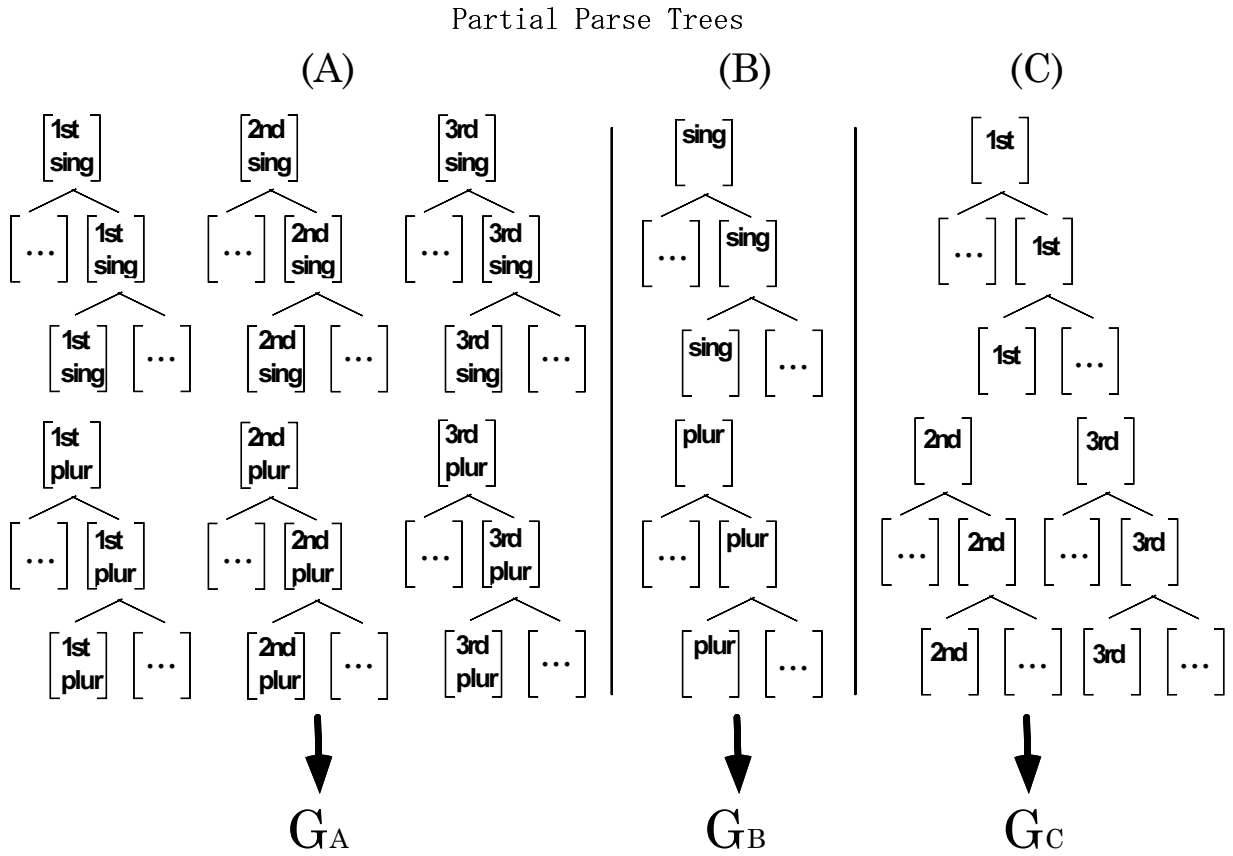

Figure 3: A sketch of compilation

other words, the features that are eliminated in the compilation to CFG (c) is a subset of the features removed in the compilation to the other two CFGs. Table 1 shows the numbers of the nonterminals, the rewriting rules in the CFGs, and the eliminated features in the compilation. Note that the intersection of the eliminated features of CFG (a) and CFG (b) equals to those of CFG (c).

We measured the parsing time of our parser with CFG filtering with the following settings. The parser is an extended version of the TNT parser [3].

1. CFG (a)
2. CFG (b)
3. CFG (a) and CFG (b) combined
4. CFG (c)

Note that in our experiments, we achieved maximal parsing speed by the original CFG filtering where we use only CFG (c) . The setting that we use both of CFG (a) and CFG (b) achieved the maximal speed in our multiple CFG filtering.

We extracted 500 sentences from the ATIS corpus [9] for the experiments. The average sentence length is 7.4 words. We ran our parser on a Sun Ultra Enterprise(Ultra Sparc II 336MHz, Solaris 7, 6 GBytes memory). The average parsing time per a sentence is listed in Table 2. In the case that we use the combination of CFG (a) and CFG (b), the parsing speed was almost the same as that of CFG (c). Parsing only with CFG (a) was 4.7 times slower than the parsing with CFG (c). In the case of CFG (b), parsing was 3.5 times slower than that with CFG (c).

It should be noted that the processing time of Phase 1 with CFG (a)+(b) is less than that with

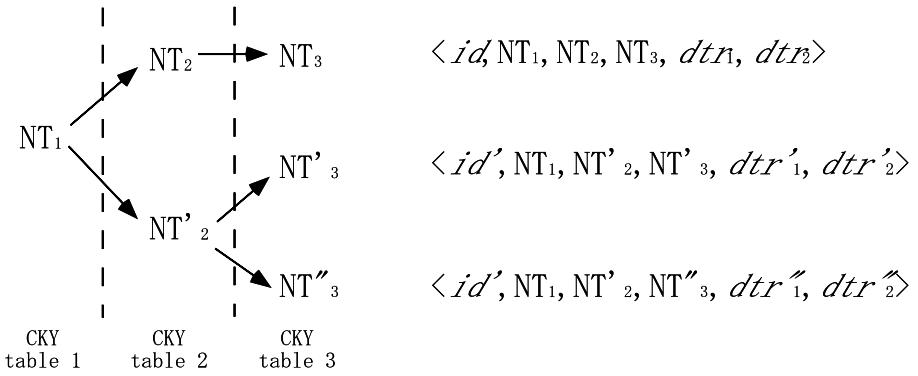

Figure 4: An example of factoring

CFG	the number of the edges
(a)	7461
(b)	19567
(b)(& (a))	2947
(c)	4711

Table 3: Comparison of the number of the edges generated in Phase 1

CFG (b), according to Table 2. Table 3 shows the number of the edges generated at Phase 1[3] . According to the table, it can be said that CFG (a) filters out the unnecessary edges which CFG (b) does not eliminate.

Let us proceed to the analysis of memory utilization. Table 2 shows the comparison of actual memory usage. It shows that, when we use the combination of CFG (a) and CFG (b), the occupied memory space was only 57% of that with CFG (c). Thus, multiple CFG filtering could effectively reduce the memory space required for CFG filtering. The space complexity of Phase 1 is higher than that of Phase 2 because Phase 1 requires the memory space to store rewriting rules. Therefore, it is effective to reduce the memory space for CFG filtering.

Note that most of these memory space is occupied by rewriting rules in CFGs. In order to achieve optimal parsing speed at Phase 1, we stored rewriting rules in a two dimensional array indexed by two nonterminals in the right hand side of the rewriting rules. Although this requires rather large memory space (proportional to the square of the number of nonterminals), other methods such as hash-tables could not achieve optimal parsing speed since parsing speed is extremely sensitive to the speed of access to rewriting rules.

[3] Factoring is ignored.

5 Conclusion

We have presented a method to improve space efficiency of a parsing method for HPSG, and reported the experimental results showing that the method could reduce memory usage with little loss of parsing performance. Currently, our group is trying to improve the parsing performance of several distinct HPSG-based grammars, including LinGO [10] and RenTAL [11]. The presented technique will contribute to the achievement of practical parsing performance for those grammars.

References

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

[2] Hiroshi Kanayama, Kentaro Torisawa, Yutaka Mitsuishi, and Jun'ichi Tsujii. A hybrid Japanese parser with hand-crafted grammar and statistics. In *COLING'2000 Proceedings*, pages 411–417, August 2000.

[3] Kentaro Torisawa, Kenji Nishida, Yusuke Miyao, and Jun'ichi Tsujii. An HPSG parser with CFG filtering. *Journal of Natural Language Engineering Vol 6(1)*, pages 63–80, 2000.

[4] Bernd Kiefer and Hans-Ulrich Krieger. A context-free approximation of head-driven phrase structure grammar. In *The Proceedings of International Workshop on Parsing Technologies*, 2000.

[5] Kenji Nishida, Kentaro Torisawa, and Tsujii Jun'ichi. An efficient HPSG parsing algorithm with array unification. In *Proceedings of the Natural Language Processing Pacific Rim Symposium 1999 (NLPRS'99)*, pages 144–149, 1999.

[6] Stuart C. Shieber. Using restriction to extend parsing algorithms for complex feature based formalisms. In *Proceedings 23rd Annual Meeting of the Association for Computational Linguistics*, pages 142–152, 1985.

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

[8] Yuka Tateisi, Kentaro Torisawa, Yusuke Miyao, and Jun'ichi Tsujii. Translating XTAG english grammar to HPSG. In *Proceedings of Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+4)*, pages 172–175, 1998.

[9] C.T. Hemphill, J.J. Godfrey, and G.R. Doddington. The ATIS Spoken Language Systems, pilot Corpus. In *Proceedings of 3rd DARPA Workshop on Speech and Natural Language*, pages 102–108, June 1990.

[10] Daniel P. Flickinger and Ivan A. Sag. Linguistic Grammars Online. A multi-purpose broad-coverage computational grammar of English. In *CSLI Bulletin 1999*, pages 64–68, Stanford, CA, 1998. CSLI Publications.

[11] Naoki Yoshinaga, Yusuke Miyao, Kentaro Torisawa, and Jun'ichi Tsujii. FB-LTAG kara HPSG heno bunpou henkan (conversion from FB-LTAG to HPSG). In *Proceedings of the Sixth Annual Meeting of The Association for Natural Language Processing*, pages 183–186, 2000. (in Japanese).