LTAG 文法による HPSG パーザを用いた構文解析

吉永 直樹† 宮尾 祐介† 鳥澤 健太郎†‡ 辻井 潤一†
†東京大学大学院理学系研究科情報科学専攻 ‡さきがけ研究 21
{yoshinag, yusuke, torisawa, tsujii}@is.s.u-tokyo.ac.jp

1 はじめに

本稿では Lexicalized Tree Adjoining Grammar (LTAG) [16, 17] 文法を使用した構文解析を, Head-driven Phrase Structure Grammar (HPSG) [8] パーザを利用して高速に行う手法について報告する. LTAG は, 幾つかの言語について大規模な文法が開発されている [12] が, 既存の LTAG パーザでは現実のテキストを解析するのに十分な速度が出ていない [1]. 本研究では, LTAG 文法を HPSG スタイルの文法に変換し, 得られた文法を用いた HPSG パーザによる構文解析が LTAG パーザでの構文解析に比べ遥かに高速であることを示す. それにより, LTAG 文法による構文解析に新しい選択肢を提供する.

LTAG から HPSG への変換手法としては, 既に我々が提案した手法 [18] があるが, 適用可能な LTAG 文法に強い制限があった. 本研究では, 任意の LTAG 文法を, 制限を満たす LTAG 文法に変換することで, HPSG スタイルの文法に完全に変換することを可能にする. 同時に我々の変換手法は, 変換で得られた文法と元の LTAG 文法の間で強い等価性を保持する. すなわち, 両文法は同じ文に対して対応の取れる導出木 (導出過程木) を生成する. これにより, 元の LTAG 文法による構文解析結果を, HPSG パーザにより得ることが可能となった.

このような, 強い意味での等価性を保つ文法変換としては, Schabes らによる CFG から Tree Insertion Grammar (TIG) への変換がある [11]. この研究では CFG を等価な TIG に変換することにより, 5 ~ 10 倍の高速化が達成されたと報告している. CFG から TIG への変換が, 構文解析の最悪計算量に関して同じ $O(n^3)$ である枠組間での変換であったのに対し, 本稿で述べる文法変換は LTAG ($O(n^6)$) から HPSG ($O(2^n)$) と, 計算量的には不利な枠組みへの変換であるため, 解析速度の高速化はそれほど自明ではない.

本研究では, 拡張された変換アルゴリズムを用いて大規模な LTAG 文法である XTAG 英語文法 [12] を HPSG スタイルの文法に変換し, 両文法に強意の等価性があることを実験結果から確認した. 実験からさらに, 得られた HPSG スタイルの文法による高速な HPSG パーザ [6, 7, 13] を用いた構文解析が, 元の XTAG 文法による構文解析よりも約 20 倍高速であることを示し, 解析速度の高速化の要因についても考察する.

2 LTAG から HPSG への文法変換

本節では変換の対象である LTAG 文法, 及び (変換後の) HPSG スタイルの文法について概説した後, 我々が以前提案した変換アルゴリズムについて述べる.

LTAG [16, 17] は **elementary tree** と呼ばれる木構造を代入 **(substitution)** と接合 **(adjunction)** と呼ばれる結合操作により組み合わせることで文解析を行う

[1] コーパスから自動学習した文法素性を含まない LTAG 文法に関しては, 実用的な速度の構文解析実験が報告されている [10].

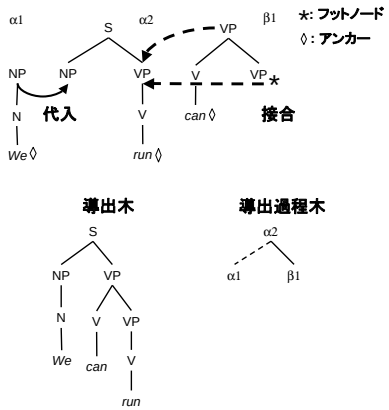

図 1: LTAG 文法における代入・接合による構文木の導出

文法記述枠組である (図 1). 各 elementary tree は語 (アンカー) を必ず一つ以上含み, LTAG における語彙項目に相当する. 代入は CFG における書き換え規則に相当するもので, elementary tree の葉ノードを他の elementary tree で置き換える操作である. 一方接合は, 根ノードと同じ記号 x でラベルづけされた特別な葉ノード (フットノード) を持つ elementary tree を, 同じ記号 x を持つノードに割り込ませる操作である. LTAG による構文解析は, 文に対して導出木 (構文木) だけではなく, 導出過程木を与えるという点に特徴がある. 導出過程木は導出木の代入と接合による導出の過程を表す. さらに本稿では, 次の 2 つの概念を導入する. elementary tree において, アンカーから根ノードに至るパスをトランク, そのパス上のノードをトランクノードと呼ぶ [2].

一方, HPSG スタイルの文法とは, [8] で提案された HPSG が採用している計算の枠組を持つ文法を指す. 本稿では HPSG スタイルの文法を以下のように定義する. 文法は型付き素性構造 [1] をデータ構造として語彙項目と文法規則とで記述される. 前者は下位範疇化要素や語彙範疇などの語特有の情報を表現する. 後者は導出木中の母娘間の一般的な文法的関係を記述し, 具体的な文法素性の値の指定は行わない.

これらの定義に従い, 我々は以前に以下のような文法変換アルゴリズムを提案した [18].

語彙項目の変換 LTAG の語彙項目である木構造を HPSG の語彙項目である素性構造に変換し,

文法規則の定義 LTAG の代入と接合に対応する, 変換された素性構造を組み上げる文法規則を定義する.

しかし, 語彙項目の変換では, HPSG の語彙項目に一対一に対応づけられる elementary tree のみを対象としていた. 具体的には以下の 2 つの条件を満たす木のみ変換可能であった.

条件 **1** アンカーの数が 1 つ

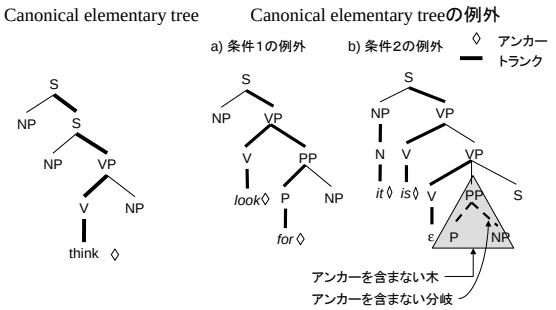

図 2: Canonical elementary tree とその例外

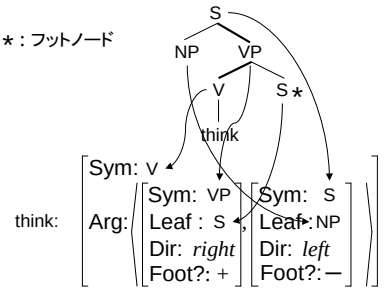

図 3: Elementary tree から HPSG の語彙項目への変換

条件 **2** 全ての分岐にトランクノードが含まれる

本稿ではこれらを満たす木構造を **canonical elementary tree** と呼び, トランクノードが含まれない分岐で子が全て葉ノードのものをアンカーを含まない分岐と呼ぶ. トランクノードを含まない深さが 2 段以上の部分木をアンカーを含まない木と呼ぶ (図 2) .

図 3 が変換の例を示している. 条件 1, 2 を満たす木構造は, 各トランクノードの姉妹ノードとなる葉ノードを, 下位範疇化要素としてスタックに格納することで, HPSG の素性構造に変換される (図中 Arg 素性) . 代入と接合に当たる操作は, 前者はスタックを一つずつ消費して単一化可能なノードを下位範疇化する文法規則, 後者はスタックを繋ぎ合わせる文法規則で模倣することができる.

この変換が強意の文法変換であるためには, 変換後の文法において, 元の文法の各葉ノードに代入・接合可能な木に対応する素性構造のみが下位範疇化されなくてはならない. そこで, 我々は各分岐を親ノードの記号, 葉ノードの記号, トランク上のノードから見た葉ノードの位置, 葉ノードがフットノードであるかどうか, という 4 つ組で表現し, 文法規則適用時にこれらがチェックされるようにした (図中 Sym 素性, Leaf 素性, Dir 素性, Foot? 素性) . HPSG の構文木生成時の文法規則適用の履歴を辿ることで, LTAG の導出過程木における木同士の結び付きを復元することができるので, 強い等価性が保たれる.

3 変換アルゴリズムの拡張

本稿では任意の LTAG 文法に対して前節で定義した変換アルゴリズムを適用するために, 条件 1, 2 を満たさない elementary tree を canonical elementary tree に変換するアルゴリズムを提案する.

3.1 条件 1 のみを満たさない木の変換

図 2 中央の例のような, 条件 1 に違反する木の変換について述べる. 複数のアンカーを持つ elementary tree は複数の canonical elementary tree に分割し (図 4) , これにより分割された木を HPSG の複数の語彙項目に変換することができる.

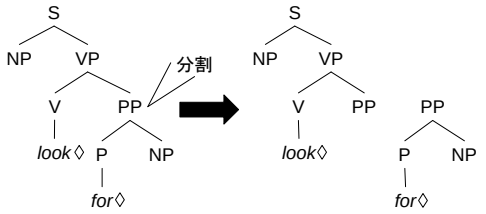

図 4: 複数のアンカーを持つ elementary tree の分割

```
procedure divide_tree_into_subtrees(MT)
begin
  ST := φ
  if (number(MT) = 1)
    return {MT}
  else
    A := select(MT)
    ⟨ST, T⟩ := divtree(MT, A)                          ... (1)
    foreach T (T)
      SST := divide_tree_into_subtrees(T)
      ST := ST ∪ SST
    end foreach
    ST := ST ∪ {ST}
    return ST;
  end if
end

procedure divtree(MT, A)
begin
  T := φ
  for i := 1 to depth(MT, A)-1
    if (nonleaf(arg(trunk(i))))
      ⟨MT', T⟩ := cut(MT, arg(trunk(i)))               ... (2)
      address(trunk(i), Address)
      mark(Address, MT', T)                            ... (3)
      T = T ∪ T
      MT := MT'
    end if
  end for
  ST := MT
  return ⟨ST, T⟩
end

number:   アンカーの数を返す
select:   アンカーを 1 つ返す
depth:    アンカーの木の中の深さ
trunk:    深さ i のトランクノードを返す
arg:      トランクノードの姉妹ノードを返す
cut:      トランクノードの姉妹ノードを切断して,
          姉妹ノードが根ノードとなる部分木を返す
nonleaf:  ノードが葉ノードであれば 1 を返す
address:  elementary tree 中のノードのアドレスを返す
mark:     アドレスを切断されたノードに記録する
```

図 5: 複数のアンカーを含む elementary tree MT から 1 つのアンカーを含む複数の部分木 ST への変換

図 5 に示す関数 `divide_tree_into_subtrees` が複数のアンカーを持つ elementary tree MT を複数の canonical elementary tree ST に変換する分割アルゴリズムである. まず最初にアンカー A が選ばれ, アンカー A から組み上げることができる canonical elementary tree ST が抜き出される (図 5 (1)). 根ノードからアンカー A に至るパスをチェックして姉妹ノードが葉ノードでなく別のトランク上にあれば, そこでその姉妹ノードを根ノードとするような部分木を切り取る (図 5(2)) .

ここで注意しておかなければならないのは, 切断されたノードは代入を模倣する文法規則で組み上げられること, また切断された木同士のみが代入の対象となるということである. このため我々は分割点のノードに木構造中のアドレスを記録するようにした (図 5(3)) . 代入を模倣する文法規則は, 記号と同時にこのアドレスをチェックするように拡張される. これによりこの分割点は異なる elementary tree の代入・接合とは区別されるため強意の等価性は保存される.

ところでどのような順序で関数 `select` がアンカー A を選択するかという問題については議論の余地がある. 複数のアンカーを持つ木の多くは “kick the bucket” や “in front of” のようなイディオムを表現している. つまり, この問題はイディオムにおいてどのアンカーが最も重要な統語役割を果すかという問題に帰着する. 本来は投射などの言語学的な概念を用いて解くべきだが, 構文解析の観点からは選択の順序に関わらず等価性は保たれ

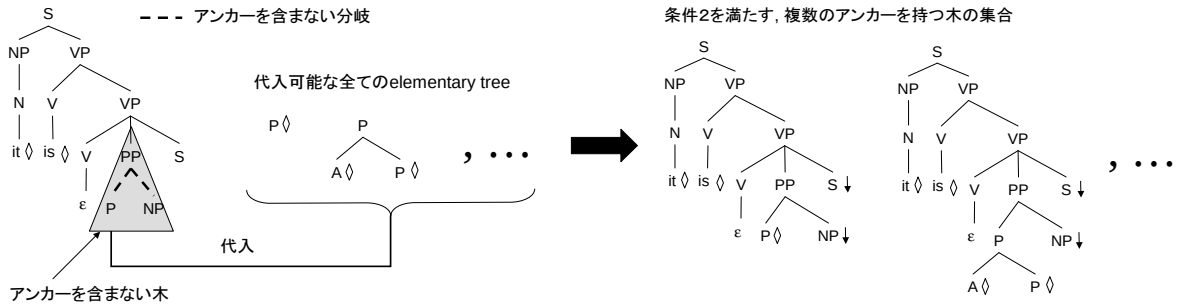

図 6: アンカーを含まない分岐への off-line substitution の適用

```
procedure expand_tree_into_anchored_tree(NT)
begin
  BR := na_br(NT)
  MT := {NT}
  foreach BR (BR)
    S := select(BR)                                   ... (1)
    IT := initial(S)                                  ... (2)
    MMT := φ
    foreach MT (MT)
      TMT := substitute(MT, S, IT)                    ... (3)
      MMT := TMT ∩ MMT
    end foreach
    MT := MMT
  end foreach
  return MT
end

na_br:      アンカーを含まない分岐を全て返す
initial:    葉ノードに代入可能な elementary tree を全て返す
select:     葉ノードを 1 つ返す
substitute: 葉ノードに off-line substitution を行い,
            条件 2 を満たす木の集合を返す
```

図 7: 条件 2 に違反する elementary tree NT を条件 2 を満たす elementary tree の集合 $\mathcal{MT}$に変換するアルゴリズム

るため, 現在の実装では単に一番左のアンカーから順に選ぶようにしている.

3.2 条件 2 を満たさない木の変換

この節では, 図 2 右の例のような, アンカーを含まない木を, アンカーを含まない分岐の子の葉ノードに他の elementary tree を代入させることでアンカーを含む部分木に変換するアルゴリズムを提案する (図 6). 我々はこの変換を *off-line substitution* と呼ぶ. この変換により, アンカーを含まない木を持つ elementary tree を変換することが可能となる.

図 7 の関数 `expand_tree_into_anchored_tree` が, アンカーを含まない木 NT と他の elementary tree $\mathcal{MT}$ を結合させるアルゴリズムを示している. まずアンカーを含まない分岐から葉ノード S を一つ選ぶ. その葉ノード S に off-line substitution を施す (図 7 (1)) . 関数 `substitute` で off-line substitution を行う (図 7(3)) .

ここでこのアルゴリズムの停止性について述べておく必要がある. 関数 `initial` は全ての代入可能な elementary tree を返す (図 7(2)) ので, その elementary tree の中には自分自身, つまりこのアルゴリズムを適用すべき elementary tree が含まれる可能性がある. これにより off-line substitution の適用が停止しない可能性があるが, これは単にアンカーを含まない木を予め切断しておけば良い. 具体的には木を分割するアルゴリズムを, 条件 2 を満たさない elementary tree にも適用し, アンカーを含まない木を切り出して, off-line substitution を施す木の候補から外す.

どの葉ノードを代入するノードとして選ぶかに関してはやはり議論の余地がある. 条件 2 を満たさない elementary tree は *it-clefts* や *equative be* など, 下位範疇化する要素だけでなくその子孫の木構造を指定するような, 直接支配を越えた構文構造に対応する. この問題はどの葉ノードが最も重要な統語役割を果すかという問題に帰着する. 前節と同様に, 言語的な概念を元に解くべきであるが, 構文解析の観点からは支障がないため現在の実装では最も左の葉ノードに対して代入することとする.

4 評価実験及び考察

本稿で提案したアルゴリズムを, 素性構造記述言語 LiLFeS [4] で実装し[2], ペンシルバニア大学で開発されている XTAG 英語文法 [12] の最新版[3]を変換した. 変換された XTAG 英語文法及び変換後の HPSG 文法の詳細を表 1 に示す[4]. 以下の実験は全て Pentium III Xeon/700MHz (4GB メモリ) 上で行われた.

次に, LTAG パーザと HPSG パーザを用いて構文解析速度の比較実験を行った. 比較に用いたパーザは, ペンシルバニア大学の LTAG パーザ (lem) [10] と, 我々の研究室で開発している HPSG パーザ (TNT) [13] である. 前者は LTAG において最も高速な構文解析アルゴリズムの一つである head-corner パージング [15] を実装したパーザである. 後者は HPSG 文法から予めコンパイルされた CFG により生成し得る構文木候補を絞った後, HPSG の構文解析を行う 2 段階の構文解析アルゴリズム [14, 3] を実装したパーザである. 前者は C, 後者は C++で実装されている. テストコーパスは Penn Treebank の ATIS コーパス [5] から 20 語以下の文 572 文のうち先頭 100 文から 70 文 (平均語長 8.0 語) を抜き出したもの用いた[5]. 表 2 に構文解析時間の平均値を示す. 表から分かるように, LTAG パーザに対して, HPSG パーザの方が約 19.9 倍高速であった.

LTAG パーザに対して HPSG パーザが高速である要因を検証する. 比較に用いた HPSG パーザは, HPSG 文法を近似する CFG を使って構文木の候補を絞る. すなわち変換で得られた HPSG 文法の多くの部分が CFG により近似できていれば, 経験的な計算量は CFG の $O(n^3)$ に近づき TAG の $O(n^6)$ を下回る. 今回の変換対象である XTAG 英語文法は elementary tree のうち 99%が CFG と等価な生成能力を持つ Tree Insertion Grammar の枠

[2]本プログラムは下記の URL で公開している.
http://www-tsujii.is.s.u-tokyo.ac.jp/rental/

[3]後述する構文解析速度の比較実験に用いた LTAG パーザ [10](ftp://ftp.cis.upenn.edu/pub/xtag/lem/lem-0.13.0.tgz) に付属する文法を使用した.

[4]表中の数字は, 実際には elementary tree のテンプレートの数である. なお, 特定の木構造に対して LTAG パーザが正しく解を生成しない場合があるので, そのような木 32 種類を除いたものを使用している.

[5]この数字は LTAG パーザが時間切れした 17 文と, LTAG パーザの前処理系のバグ等により解析結果が合わない 13 文を除いたものである.

表 1: 条件 1, 2 による XTAG 英語文法 (LTAG) の elementary tree の分類とその数, 及びそれぞれの分類に対応する得られた HPSG 文法 (HPSG) の語彙項目の数

文法	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$	計
LTAG	326	764	54	50	1,194
HPSG	326	1,992	1,083	2,474	5,875

$\mathcal{A}$: canonical elementary trees
$\mathcal{B}$: 条件 1 のみに違反する elementary tree
$\mathcal{C}$: 条件 2 のみに違反する elementary tree
$\mathcal{D}$: 条件 1, 2 に違反する elementary tree

表 2: LTAG パーザ (lem) と HPSG パーザ (TNT) の ATIS コーパス 70 文に対する構文解析速度の比較

パーザ	解析時間 (秒)
lem	32.1
TNT	1.6

組内で記述できることが報告されており [11][6], CFG で大幅に構文木の候補を絞ることができると考えられる.

LTAG 文法を近似する CFG を使って構文解析を行う手法として, TAG から直接 CFG を抜き出す手法も提案されているが [9], 我々の HPSG パーザを利用した手法の方が高効率であると考える. 彼らの手法による構文解析速度の実験結果は現時点では示されていないが, 彼らの手法が基本的に elementary tree 中のノードを CFG の非終端記号としているのに対し, HPSG 文法では非終端記号は素性構造に対応する. 本稿の変換においては, 変換後の文法の語彙項目の素性構造に元の文法の elementary tree の全てのノードがエンコードされているため, 近似された CFG の非終端記号はノードが持つ記号だけでなく, 長距離の依存関係も含んだ木構造全体の情報を持っている. これは, 非終端記号の数が 7552 (CFG 規則は 54595) と, ノードの記号の数 19 に対して 400 倍近いことからも裏付けられる. そのため, LTAG を CFG で近似する手法より効率的に構文木候補を絞り込めるため, 報告したような高速化が達成されたと考える.

5 まとめと今後の課題

本稿では, LTAG から HPSG への強意の等価性を保つ文法変換アルゴリズムを拡張し, 任意の LTAG 文法を HPSG スタイルの文法に変換することを可能にした. これにより, HPSG パーザを用いて, LTAG の構文解析結果を得ることを可能にした. 実験では, LTAG の大規模文法である, XTAG 英語文法を完全に変換した. 変換された文法を用いて, LTAG パーザと HPSG パーザの構文解析速度の比較実験を行い, HPSG パーザが LTAG パーザに比べて約 20 倍高速であることを確認した.

今後の課題としては, 更に高速な構文解析手法 [7] の導入, 素性構造上の確率モデル [19] の導入があげられる.

謝辞

LTAG パーザと HPSG パーザの比較実験のために, LTAG パーザの使用を快諾してくださったペンシルバニア大学の Anoop Sarkar 氏に心より感謝します.

[6]ただし 7 年前のバージョンでのデータである.

参考文献

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

[2] Robert Kasper, Bernd Kiefer, Klaus Netter, and K. Vijay-Shanker. Compilation of HPSG to TAG. In *Proc. 33rd ACL*, pages 92–99, 1995.

[3] Bernd Kiefer and Hans-Ulrich Krieger. A Context-Free approximation of Head-Driven Phrase Structure Grammar. In *Proc. 6th IWPT*, pages 135–146, 2000.

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

[5] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The Penn treebank: Annotating predicate argument structure. In *Proc. of ARPA '94*, pages 114–119, 1994.

[6] Yusuke Miyao, Takaki Makino, Kentaro Torisawa, and Jun'ichi Tsujii. The LiLFeS abstract machine and its evaluation with the LinGO grammar. *Natural Language Engineering Special Issue – Efficient Processing with HPSG: Methods, Systems, Evaluation*, 6(1):47–61, 2000.

[7] Kenji Nishida, Kentaro Torisawa, and Jun'ichi Tsujii. An efficient HPSG parsing algorithm with array unification. In *Proc. 5th NLPRS*, pages 144–149, 1999.

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

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

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

[11] Yves Schabes and Richard C. Waters. Tree Insertion Grammar: A cubic-time parsable formalism that lexicalizes Context-Free Grammar without changing the tree produced. *Computational Linguistics*, 21(4):479–513, 1995.

[12] The XTAG Research Group. A Lexicalized Tree Adjoining Grammar for English. http://www.cis.upenn.edu/~xtag/, 1999.

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

[14] Kentaro Torisawa and Jun'ichi Tsujii. Computing phrasal-signs in HPSG prior to parsing. In *Proc. 16th COLING*, pages 949–955, 1996.

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

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

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

[18] 吉永 直樹, 宮尾 祐介, 建石 由佳, 鳥澤 健太郎, 辻井 潤一. FB-LTAG から HPSG への文法変換. 第 6 回言語処理学会年次大会発表論文集, pages 183–186, 2000.

[19] 宮尾 祐介, 辻井 潤一. 確率付き項構造による曖昧性解消. 言語処理学会第 6 回年次大会発表論文集, pages 495–498, 2000.