NAIST-IS-MT1251118

修士論文

依存構造解析器への並列構造解析の組み込み

吉本 暁文

2014 年 3 月 16 日

奈良先端科学技術大学院大学
情報科学研究科 情報処理学専攻

本論文は奈良先端科学技術大学院大学情報科学研究科に
修士 (工学) 授与の要件として提出した修士論文である。

吉本 暁文

審査委員：

松本　裕治	教授	（主指導教員）
中村 哲	教授	（副指導教員）
新保　仁	准教授	（副指導教員）
Kevin Duh	助教	（副指導教員）

依存構造解析器への並列構造解析の組み込み*

吉本 暁文

内容梗概

構文解析器で捉えることが難しい言語現象の一つに並列構造がある．並列構造には，一般的な依存構造や構成素構造の解析における統計的な特徴からは捉えられない特有の傾向があるため，これを従来の解析と分けて捉えられる仕組みを用意しなければならない．これまでに，並列構造を扱うアルゴリズムと，それを一般的な構文解析器と組み合わせる方法が提案されてきた．しかし，構文解析の過程で並列構造の特徴を考慮させる手法には至っていない．依存構造解析において，構文解析中に並列構造の特徴を考慮させることで，構文解析，並列構造解析双方の精度を向上させられると予想できる．本論文では，Eisner による依存構造解析規則に，並列構造解析のための専用規則を追加することを提案する．

キーワード

依存構造解析, 並列構造解析, Eisner アルゴリズム

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 修士論文, NAIST-IS-MT1251118, 2014 年 3 月 16 日.

Incorporating Coordinate Structure Analysis into a Dependency Parser*

Akifumi Yoshimoto

Abstract

Parsing coordinate structure is difficult even for state-of-the-art parsers. It has been found that the similarity of coordinating conjuncts provides useful information in determining the scope of coordination, and some specialized algorithms exist for coordination structure analysis on the basis of this similarity. However, these are specialized algorithms for coordination structure analysis, and are designed independent of parsers.

On the other hand, Eisner's dependency parsing algorithm builds syntactic dependency trees in a way that computation of conjunct similarity is inherently difficult. In this paper, we propose a set of parsing rules that augment Eisner's, so that the similarity of coordinating conjuncts can be computed. These rules could improve the accuracy of the dependency parser on coordinate structure.

Keywords:

Dependency Parsing, Coordinate Structure Analysis, Eisner Algorithm

*Master's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1251118, March 16, 2014.

謝 辞

本研究を進めるにあたり，主指導教官としてご指導，ご助言を頂いた自然言語処理学研究室の松本裕治教授に深く感謝いたします．

また，副指導教官としてご指導，ご助言を頂いた知能コミュニケーション研究室の中村哲教授に深く感謝いたします．

自然言語処理学研究室の新保仁准教授には，副指導教官としてのご指導，ご助言はもちろん，プログラムコードの書き方や論文の書き方，考え方に至るまで大変幅広くご指導を頂きました．また，本研究のみならず，研究室の計算機資源の管理に関する作業について，どんな困難があった時にもいつも尽力して下さりました．ここに深く感謝いたします．

自然言語処理学研究室の Kevin Duh 助教には，本研究に関するご指導，ご助言を頂いたほか，関連する興味深い研究の話をして頂いたり，関連する研究で利用するための計算機資源の構築を一緒に企画して実行したり，広い視野で接して頂いたりする中で，多くの影響を受けました．ここに深く感謝いたします．

国立遺伝学研究所生命情報研究センター遺伝子発現解析研究室特任研究員の原一夫氏には，本研究のために静岡から奈良まで幾度も往復して議論して頂き，大変多くのご助言を頂きましたことを深く感謝いたします．

NTT コミュニケーション科学基礎研究所の林克彦氏には構文解析の考え方を教えて頂いたり，本研究を始める以前にも相談させて頂いたりしたことを深く感謝いたします．

自然言語処理学研究室の近藤修平氏には構文解析の考え方を教えて頂いたり，関連論文を教えて頂いたりしたことを深く感謝いたします．

自然言語処理学研究室の同期の方々，特に実装について競争させて頂いたり教えて頂いたりした光瀬智哉氏，発表のしかたや資料の修正に関して多くのご助言を頂いた椿真史氏，三澤賢祐氏，吉本一平氏，計算機の特性について議論させて頂いた和良品友大氏に深く感謝いたします．

研究室での生活や手続きを支えて頂いた北川祐子秘書に深く感謝いたします．

研究や学生生活で状況を共有し，共に過ごしてきた自然言語処理学研究室の皆様に深く感謝いたします．

研究に安心して取り組める日常生活を支えて頂き，見守って頂いた家族の皆様に深く感謝いたします.

目 次

謝 辞 **iii**

第 1 章 序論 **2**
1.1 背景 2
1.2 本論文の貢献 4
1.3 本論文の構成 4

第 2 章 関連研究 **6**
2.1 依存構造解析 6
2.1.1 Eisner アルゴリズム 7
2.1.2 Johnson による文法表記 11
2.2 並列構造のアノテーション 15
2.3 並列構造解析 17
2.3.1 名詞句の並列構造解析 17
2.3.2 文全体を対象にした並列構造解析 17
2.3.3 並列構造解析と従来の構文解析器を利用した総合的な解析 19
2.3.4 並列構造解析で有効な素性 19

第 3 章 並列構造を扱うための規則 **22**
3.1 対象とする並列構造 22
3.2 提案規則 23
3.3 提案規則における擬似曖昧性の回避 28
3.4 提案規則の計算量 28

第 4 章 提案規則による正解構文木の作成 **34**
4.1 依存構造アノテーションから正解構文木へ変換するアルゴリズム . 34
4.2 既存の依存構造アノテーションからの正解構文木の作成 40
4.2.1 変換できなかった事例 40
4.2.2 変換された構文木に出現した規則 44

第 5 章　結論 **51**
5.1　本研究の成果 51
5.2　展望 52

図 目 次

2.1 例文 “Sandy gave the dog a bone” の依存構造 6
2.2 1st order Eisner アルゴリズム 8
2.3 Eisner アルゴリズムによる解析過程 10
2.4 図 2.1 の依存構造に対して単純な CFG を用いたときの二種類の導出 12
2.5 図 2.1 の依存構造に対して split-head の CFG を用いたときの導出 . 12
2.6 図 2.1 の依存構造に対して $O(n^3)$ 版の split-head の CFG を用いたときの導出 14
2.7 図 2.1 の依存構造に対する Eisner アルゴリズムの導出 14
2.8 複数考えられる並列項目の範囲 20
2.9 表層と意味の類似性 20
2.10 表層と意味的類似性だけでは捉えられない動詞並列の例 21

3.1 提案規則：conjunct の作成 24
3.2 提案規則：conjunct の組み合わせ 24
3.3 提案規則による解析過程 29
3.4 Eisner 規則 (1st order) による導出例 30
3.5 提案規則による並列構造の導出例 31
3.6 並列構造が他の単語の Head になる例 32
3.7 図 3.6 における並列構造の導出例 32
3.8 並列構造が入れ子になる例 33
3.9 図 3.8 における並列構造の導出例 33

4.1 状態遷移図 37
4.2 or の特殊な用法に関する事例 46
4.3 CC の依存関係の方向が左向きになる例 47
4.4 CC に対応する CONJ が存在しない例 48
4.5 CC の後に CONJ が 2 回出現する例 1 49
4.6 CC の後に CONJ が 2 回出現する例 2 50

表 目 次

1.1 Penn Treebank における並列構造の割合（Stanford Basic アノテーション） 3

2.1 1st order span 7
2.2 Eisner 規則 (1st order) 9
2.3 表 2.2 に対応する記号表記 15
2.4 並列構造の各種変換 16
2.5 [Hara et al., 2009] による並列項目の種類別の精度分析（一部） . . 20

3.1 Stanford アノテーションのための並列構造規則（文字表記） . . . 24
3.2 Stanford アノテーションのための並列構造規則（図形表記） . . . 25

4.1 提案規則の非終端記号と対応する略記 35
4.2 Stanford typed dependencies manual （抜粋） 44
4.3 変換アルゴリズムを適用した際に出現した規則 45

第1章　序論

1.1　背景

近年，ウェブ文書に代表される膨大なテキストデータから，有用な情報を得ることに対する需要が高まっている．膨大なテキストデータに対し，必要な情報を人手で探し出すことは困難なため，効率的にこの需要に応えるためには検索や翻訳などの処理を自動化することが必要である．自然言語処理は，計算機を用いてテキストデータの中から必要な情報を自動的に検索したり翻訳したりすることを目的とし，この需要に応えるものである．

自然言語は計算機言語と異なり，一般に解釈が一意に定まらず，膨大な量の解釈が存在しうる．これら複数の解釈が存在することを曖昧性があるという．これらの中から正しい解釈を見つけ出すことができなければ，必要としている検索結果や翻訳結果を得ることができない．

よって自然言語を計算機に処理させるためには，テキストの曖昧性の解消が必要となる．テキストの曖昧性には，単語の意味の曖昧性，構文的な曖昧性，照応や省略に関する曖昧性などがあるが，本論文では構文的な曖昧性の解消に焦点を当てる．

構文的な曖昧性とは，自然言語の文を構成する単語間の結びつきとその役割における曖昧性である．この曖昧性には前置詞付加や並列句による曖昧性が挙げられる．例えば前者では，“ I saw a girl with a telescope. ”という文において，“ with ”が修飾しているのが“ saw ”なのか，“ girl ”なのかによって，誰が“ telescope ”を持っているかの解釈が変わる．後者の例としては，“ I am a freshman advertising and marketing major. ”という文 [Hanamoto et al., 2012] において，“ and ”が並列している句の範囲によって解釈が変わる．

1. I am a (freshman advertising) and (marketing major).

   私は広告の一年生でマーケティング専攻です．

2. I am a freshman (advertising) and (marketing) major.

   私は広告とマーケティングが専攻の一年生です．

	wsj 2-21	wsj 22	wsj 23
文の数	39832	1700	2416
CC/CONJ タグを持つ文の数	17901 (44.9 %)	771 (45.4 %)	1035 (42.8 %)
CC タグの数	24004	1003	1384
CONJ タグの数	24367	1007	1348

表 1.1: Penn Treebank における並列構造の割合（Stanford Basic アノテーション）

こうした複数ある解釈の中でどれが正しいかを計算によって選び出力することを構文解析といい，これを実現する実装を構文解析器という．例文中で，括弧で括られた範囲は並列句を構成する単語列であり，これを並列項目と呼ぶ．現在の構文解析器では 1. のように解析される場合があるが，このように解析すると “ advertising ” も “ major ” の一つであるという解釈にならない．しかし，この例文は人が読んだときに “ advertising ” と “ marketing ” を並列していると捉えられるため，2. のように解釈されるべきである．

英語における並列句はほとんどの場合，“ and ”, “ or ” などの並列を表す表現を伴って現れる．この表現のことを並列マーカ（あるいは並列キー）と呼び，並列マーカによって並列される表現は多くの場合，表層的な類似性や意味的な類似性を持つ．上の例文では，“ advertising ” と “ marketing ” はどちらも “ -ing ” を接尾辞に持つという表層的な類似性を解析の手がかりに利用することができると考えられる．

構文解析の手法は大きく分けて二種類あり，単語間の関係による解析を依存構造解析と呼び，句の範囲とその再帰的な構造による解析を句構造解析と呼ぶ．依存構造解析は固有の言語や語順に依存しないため，関係抽出 [Nguyen et al., 2009] や機械翻訳 [Katz-Brown et al., Xie et al., 2011, 2011] といった幅広い応用分野に利用されている．並列構造では並列項目となる単語列の範囲における曖昧性を解消する必要があるが，依存構造解析では句のような単語列の範囲に相当するものが定義されないために，他の依存関係の同定に比べて特に並列関係の同定を行うことが難しい．

しかし並列構造の自然言語の文における出現率は高い．例えば Penn Treebank の WSJ セクションでは，表 1.1 に示すように，4 割程度の文で並列構造が出現する．この表において CC タグは “ and ”, “ or ” などの等位接続詞を示すアノテーションであり，CONJ タグは並列される項目を示すアノテーションである．

並列関係の曖昧性を解消する手法として Hara らによる系列アライメントを用いた並列句の同定手法 [Hara and Shimbo, Hara et al., 2007, 2009] がある．Hara らは

さらにこの手法が従来の構文解析器と相補的に働く可能性を示した．

よって依存構造解析において，並列関係の曖昧性を解消する手法を組み込んだ統合的な解析器を作ることで，並列句を含む文の依存構造解析の精度を向上できると考えられる．

しかし，英語の依存構造と並列構造の両方を直接解く手法は未だ提案されていない．句構造解析については，同時に扱う代わりに，句構造解析と並列構造解析の解析結果を一致させるように最適化する手法である双対分解を用いた解析器が [Hanamoto et al., 2012] によって提案されている．この解析器によって，従来の構文解析と並列構造解析を組み合わせて解析精度が向上することが示された．

1.2 本論文の貢献

本研究の貢献は，以下に挙げるものである．

1. 依存構造解析の過程で並列構造を捉えるための規則の提案

   Eisner 法 [Eisner, 1996] による依存構造解析手法に追加できる規則として並列構造を扱う規則を提案した．

2. 既存の依存構造アノテーションからの機械的な変換手法の提案

   句構造から依存構造への変換の一つである Stanford アノテーション [Klein and Manning, 2002] から，提案した規則による構文木を自動的に変換するアルゴリズムを提案した．

3. 並列構造のパターンの分類

   英語の句構造アノテーションが付与された大規模なコーパスである Penn Treebank[Marcus et al., 1993] から Stanford アノテーションを用いた変換における並列構造の出現パターンを分類し，ほとんどの場合で提案したアルゴリズムによる構文木の自動変換が適用できることを示した．

1.3 本論文の構成

本論文は次の構成となっている．

1. 第 2 章では最初に依存構造解析と，その代表的なアルゴリズムについて述べる．そして並列構造が依存構造でどのように表されているか，並列構造がこれまでどのように扱われてきたかについて述べる．2 章の最後では，並列構造の特徴と並列構造を構文解析で扱うべき理由について述べる．

2. 第 3 章では本研究で提案する手法について述べる．第 2 章で述べたアルゴリズムの拡張として，追加される規則について詳細に述べる．

3. 第 4 章では，第 3 章で提案した手法を用いて構文解析器を作るために必要な正解構文木を作るためのアルゴリズムについて述べる．また，既存のコーパスに対して適用した際に，どの程度変換できるかについて述べる．

4. 第 5 章では本研究を通して得られた成果についてまとめ，今後の展望について述べる．

第2章　関連研究

2.1　依存構造解析

自然言語の文を，構成する単語の列 $W = w_1, w_2, ..., w_n$ で表したとき，$w_i \leftarrow w_j$ を依存関係と呼ぶ．ここで，w_i を子（あるいは Dependent，Modifier），w_j を親（あるいは Head）と呼ぶ．各単語 $w_i \in W$ は高々 1 つの親を持つ．この単語列 W と依存関係の集合 A からなるグラフ $D = (W, A)$ を依存構造グラフと呼ぶ．W において，$w_i \leftarrow w_j$ があるとき，$i < k < j$ であるような全ての w_k に対して w_j からのパスがあることを非交差と呼ぶ．図 2.1 は [Johnson, 2007] で用いられた例文 “ Sandy gave the dog a bone ” の依存構造であるが，この図のように非交差の依存構造グラフはエッジを交差させずに描くことができる．

依存構造解析とは，文中の単語間の依存関係を求めることにより依存構造グラフを導出することである．主要な依存構造解析の手法には Transition-based の手法や Graph-based の手法がある．Transition-based の手法は基本的には線形時間で解析できる特徴があるが，一般に大域最適解を求めることができない．これに対し，Graph-based の手法はモデルの複雑さに応じて計算量が増加するが，大域最適解を求めることができる．本論文では，依存構造解析の過程で並列関係を捉えることに焦点を当てるため，全ての可能な構文木を探索できる Graph-based の手法に着目した．

Graph-based の手法には全域木による手法や，Span-based の手法がある．Span-based の手法では文の単語列における範囲を Span と呼ぶ．この手法は隣り合う Span を連結してより大きな Span を作るボトムアップの解析法で, CKY 法 [Younger, 1967] を用いて依存構造解析を行うことができる．動的計画法により，効率的な解析が

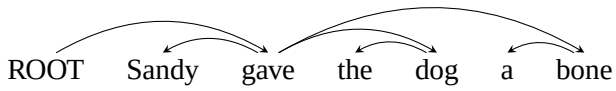


図 2.1: 例文 “Sandy gave the dog a bone” の依存構造

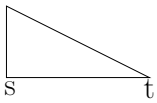
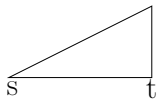
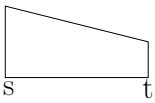
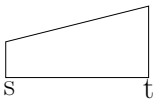
$C[s][t][\rightarrow][1]$	$C[s][t][\leftarrow][1]$	$C[s][t][\rightarrow][0]$	$C[s][t][\leftarrow][0]$
			

表 2.1: 1st order span

可能である．

並列関係の解析では，並列関係にある句の範囲の同定が重要であり，各 Span をこの句の範囲の同定に利用できると考えられるため，本研究では Span-based の手法に着目した．

2.1.1 Eisner アルゴリズム

本研究では Span-based の代表的な手法である Eisner アルゴリズムを出発点とした．本節ではこのアルゴリズムについて説明する．

Eisner アルゴリズムは注目する部分文字列の範囲を広げながらその範囲の最適な部分木を見つけていき，動的計画法のテーブルを CKY と同様に bottom-up に埋めていく．処理の各段階で注目する部分文字列の先頭の単語の位置を s とし，後尾の単語の位置を t とする．このとき，$s \leq t$ における最良の部分木を，部分木の Head における依存関係の方向 $d \in \{\leftarrow, \rightarrow\}$ と部分木の完成を示す値 $c \in \{0,1\}$ とともに保持する動的計画法のテーブル $C[s][t][d][c]$ を考える．部分木は Span で表され，$d = \leftarrow$ ならば t がその Span の Head であり，$d = \rightarrow$ ならば s が Head となる．変数 $c \in \{0,1\}$ は 1 ならば完成した Span を表し，その Span の Head の反対側にはこれ以上依存関係がないことを意味する．同様に，0 ならば未完成の Span を表し，その Span はまだ依存関係が加わって完成しなければならないことを意味する．d と c それぞれの値によって定まる合計 4 通りの Span を，それぞれに対応する図形を用いて表すと表 2.1 になる．ここで三角形は完成した Span を表し，台形は未完成の Span を表す．三角形や台形は，それぞれの単語の位置において，高い辺がある側が Head を表す．

このアルゴリズムの擬似コードを図 2.2 に示す．ここで図 2.2 の (*) で示された行を考える．この行は以下の未完成の左向き部分木の中で最大スコアを見つけるという処理になるが，これを実現するためには $s \leq r < t$ となる位置 r を，次の 2 つの完成した部分木の組み合わせが最大スコアとなるように見つけなければならない．

Initialization: $C[s][s][d][c] = 0.0 \quad \forall_{s,d,c}$
for $k \leftarrow 1 \ldots N$ **do**
  for $s \leftarrow 0 \ldots N$ **do**
    $t \leftarrow s + k$
    if $t > n$ **then**
      break
    end if

    % 未完成の Span を作る規則
    $C[s][t][\leftarrow][0] = \max_{s \le r < t}\{C[s][r][\rightarrow][1] + C[r+1][t][\leftarrow][1] + score(t,s)\}$ (*)
    $C[s][t][\rightarrow][0] = \max_{s \le r < t}\{C[s][r][\rightarrow][1] + C[r+1][t][\leftarrow][1] + score(s,t)\}$

    % 完成した Span を作る規則
    $C[s][t][\leftarrow][1] = \max_{s \le r < t}\{C[s][r][\leftarrow][1] + C[r][t][\leftarrow][0]\}$
    $C[s][t][\rightarrow][1] = \max_{s < r \le t}\{C[s][r][\rightarrow][0] + C[r][t][\rightarrow][1]\}$
  end for
end for

図 2.2: 1st order Eisner アルゴリズム

これら 2 つの完成した部分木の組み合わせに対するスコアは，組み合わせられるそれぞれの部分木のスコアの和に，単語 x_t から単語 x_s への依存関係を作るスコアを加えたものになる．こうして組み上げられた Span のスコアは，r を走査することで全ての可能な組み合わせを考慮しているため，その Span を構成する最良の部分木のスコアを反映している．文の左端に単一の root があるとすることで，文全体の最良の木は $C[1][n][\rightarrow][1]$ で表される．

初期状態は，全てのチャート上の要素はそれぞれ対応する単語のみを含む完成した部分木で，それぞれ左右に向いた三角形で表現されている．組み上げられる要素それぞれの Head の位置を h1, h2 とすると，次のステップで，h1 を h2 の Head とする未完成の部分木と h2 を h1 の Head とする未完成の部分木を作る．この要素は後の段階で完成した部分木になるよう組み上げられる．通常の CKY 解析のように，bottom-up の方法で大きい要素はより小さな要素のペアから組み上げられる．

標準的な CKY アルゴリズムで使われている方法と同様，チャート上のそれぞれの要素がその要素内の可能な部分木の中で最大のスコアを保持するように拡張することができる．文全体の Span が完成したときに，その Span を構成する最大スコアの木を復元できるようにするためにバックポインタも保持しておく．

表 2.2 のように，同一方向の未完成の Span と完成した Span を連結して完成した Span を作り，最終的に単語全てが完成した Span になるとき，解析が完了する．

解析の例を図 2.3 に示す．最初のステップではそれぞれの単語ごとに右向きと左

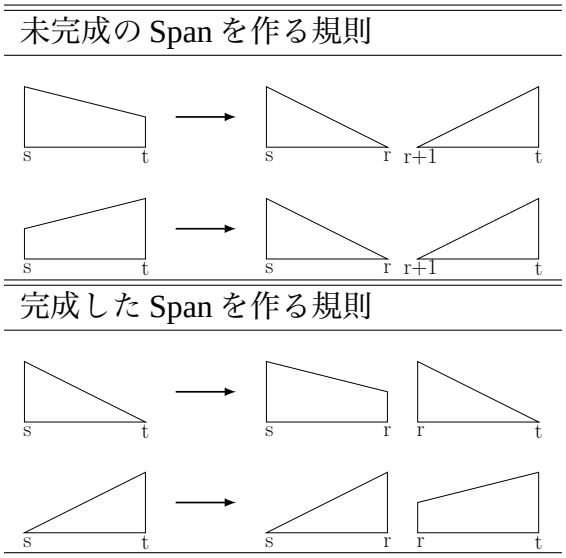


表 2.2: Eisner 規則 (1st order)

向きの完成した Span が与えられ，図ではそれぞれの向きの三角形で示している．次のステップでは隣り合う Span の組み上げを行う．

例えば，この図では to と split を組み上げて未完成の左向きの Span を作るが，これは台形で示している．このような台形は全ての向き合った三角形に対して作られ，この時それぞれの台形にはその台形に対応する依存関係の存在しやすさに相当するスコアが計算され，付与される．このスコアの計算は，台形に対応する単語列から素性を抽出し，各素性と依存関係の有無をパーセプトロンなどを用いて学習することで実現される．

次のステップで，台形と三角形を組み合わせて三角形が作られる．ここで作られた三角形には，元となった台形と三角形のスコアを足し合わせたスコアが付与される．

こうして最終的に一つの三角形ができるまで組み上げ，その中でスコアが最大となる三角形を選んだとき，その三角形を構成する Span のうち，台形があるところに依存関係があることになる．

CKY アルゴリズムを単純に変形して依存構造解析ができるように変更すると，その空間計算量と時間計算量は $O(n^5)$ となる．しかし，Eisner はそれぞれのチャート上の要素において，そのチャート上の要素の右端か左端のどちらかが Head になるようにすることで，計算量を $O(n^3)$ に抑えることができることを発見した．Eisner アルゴリズムではこれを実現するために左向きと右向きの依存関係をそれぞれ別々に解析し，これを統合していく手法となっている．こうすることで，本来 $O(n^5)$ の

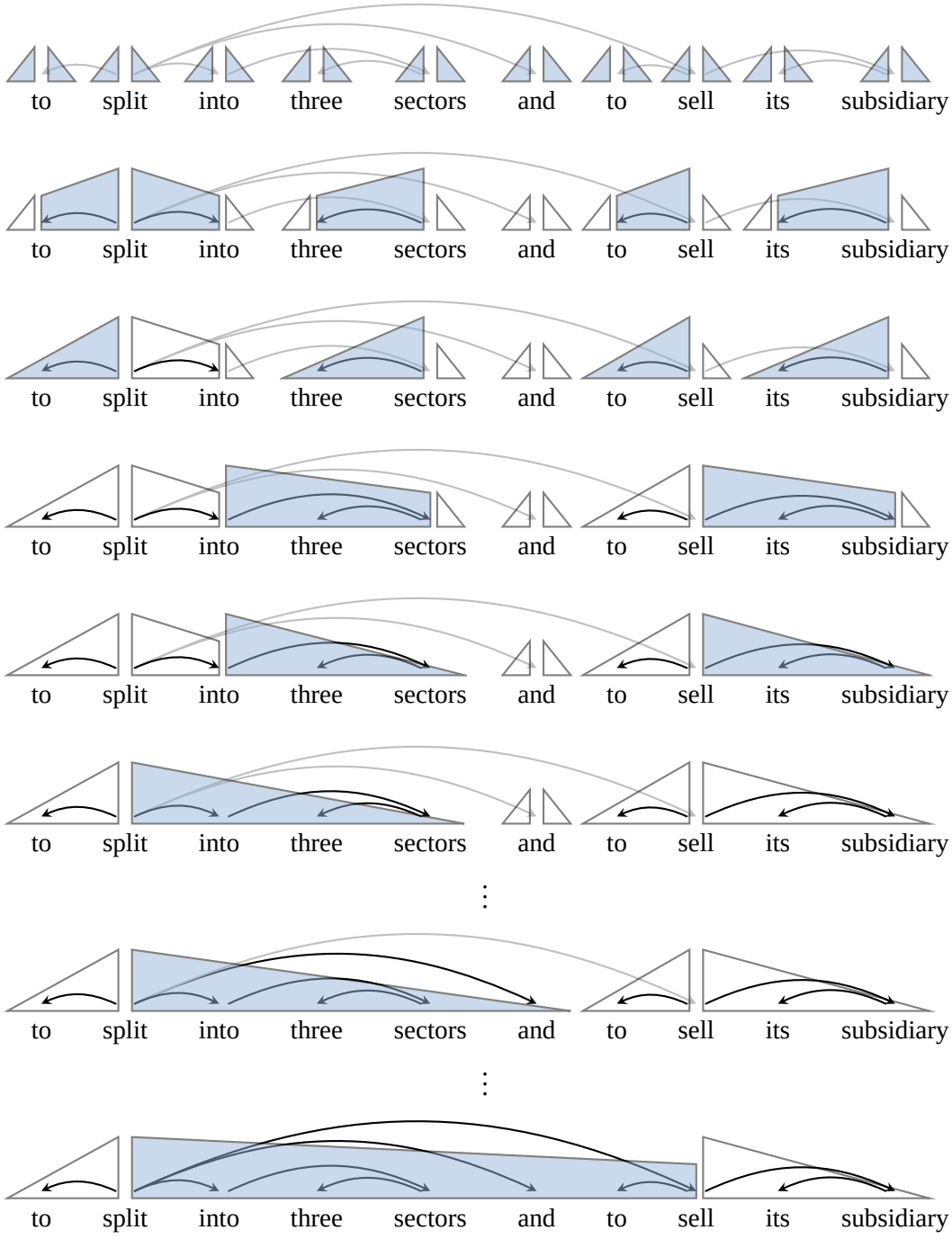


図 2.3: Eisner アルゴリズムによる解析過程

計算量であったアルゴリズムにおけるチャート上の要素の両端以外に Head の位置を別にして扱う必要がなくなり，依存関係の方向を示す 2 値の変数を加えるだけで済むようになる．このアルゴリズムの時間計算量は規則の右辺における単語の位置を示す自由変数の数に依存し，(s, r, t) がそれぞれ単語数 n に依存する自由変数であることから $O(n^3)$ となる．

2.1.2 Johnson による文法表記

Eisner アルゴリズムは三角形や台形などの図形を用いた表記の他に，対応する記号を用いて表記することができる．3 章では，複雑な文法を見通しの良い形で表記するために図形による表記の他に記号を用いた表記を用いる．ここではその表記を作る際の参考にした [Johnson, 2007] の表記を説明する．

Johnson は終端記号（単語）の表記に u, v から始まる記号を用いているが，本論文では単語の位置を表すインデックスに i, j から始まる記号を用いており，これらは同様の意味となるため，表記の一貫性を保つためにここでも同表記に統一する．

まず最も単純な CFG を考える．この文法を，非終端記号を二つ用いて表すことを考える．非終端記号に開始記号 S と，位置 i にある単語の全ての子孫を表す X_i を用いると，以下のように表記できる．

$$
\begin{array}{lcll}
\mathrm{S} & \rightarrow & \mathrm{X}_i & \text{where} \quad \overset{\mathrm{ROOT}}{\frown}\, i \\
\mathrm{X}_i & \rightarrow & i & \\
\mathrm{X}_i & \rightarrow & \mathrm{X}_j\ \mathrm{X}_i & \text{where} \quad i \overset{\frown}{} j \\
\mathrm{X}_i & \rightarrow & \mathrm{X}_i\ \mathrm{X}_j & \text{where} \quad j \overset{\frown}{} i
\end{array}
$$

それぞれの生成に割り当てられた依存関係のアノテーションは，そこで生成された部分木をどう解釈するかを示し，これによって CFG の導出を対応する依存構造に変換することができる．

しかしこの文法は，CKY 解析アルゴリズムを用いたとき，CFG の解析時間の $O(n^3)$ に Head の位置 i, j の探索時間が乗算されるため，一般的に $O(n^5)$ の計算時間がかかる．また，図 2.4 に示すように，一つの依存構造グラフが複数の CFG の導出と対応する問題がある．この問題は擬似曖昧性と呼ばれる．これは，左向きの依存関係と右向きの依存関係を任意の段階で組み合わせることができるために起こる．

擬似曖昧性を解消する方法はいくつか存在するが，その一つに “split-head” と呼ばれる方法があり，Eisner アルゴリズムでもこの方法が実現されている．

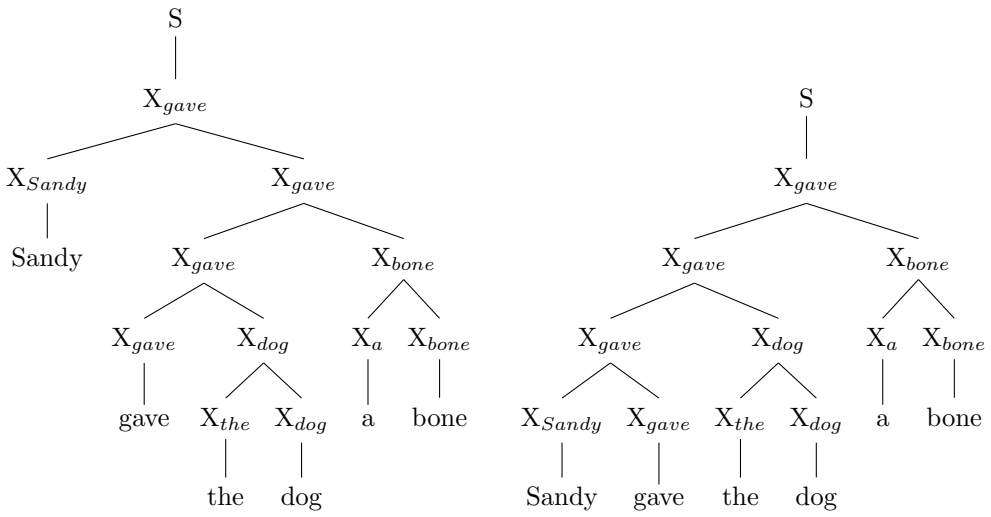


図 2.4: 図 2.1 の依存構造に対して単純な CFG を用いたときの二種類の導出

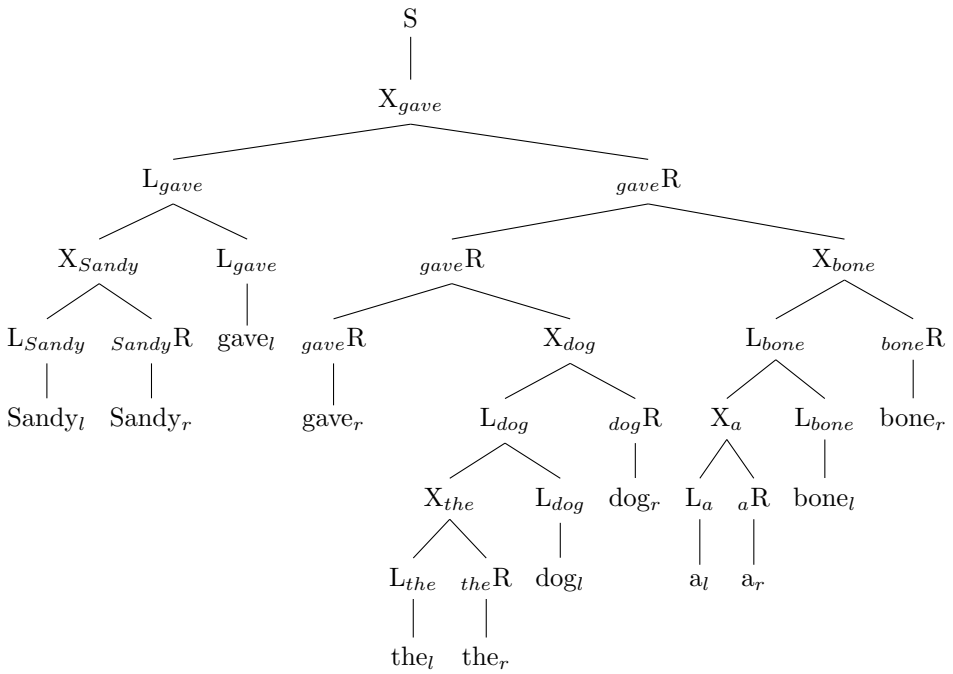


図 2.5: 図 2.1 の依存構造に対して split-head の CFG を用いたときの導出

split-head による文法ではそれぞれの位置 i にある単語を二つの独立した終端記号で表現する．ここではこれらを i_l と i_r とする．このとき，CFG を以下のように作ることができる．

$$
\begin{array}{lcll}
\mathrm{S} & \rightarrow & \mathrm{X}_i & \text{where} \quad \overset{\mathrm{ROOT}}{\frown} i \\
\mathrm{X}_i & \rightarrow & \mathrm{L}_i \; {}_i\mathrm{R} & \\
\mathrm{L}_i & \rightarrow & i_l & \\
\mathrm{L}_i & \rightarrow & \mathrm{X}_j \; \mathrm{L}_i & \text{where} \quad j \overset{\frown}{} i \\
{}_i\mathrm{R} & \rightarrow & i_r & \\
{}_i\mathrm{R} & \rightarrow & {}_i\mathrm{R} \; \mathrm{X}_j & \text{where} \quad i \overset{\frown}{} j
\end{array}
$$

この CFG による導出は図 2.5 のようになる．この CFG では，Head に対する左側と右側の依存関係を独立に組み合わせるため，擬似曖昧性が生じない．

しかしこの CFG では X_j の j で示される単語位置が被らないため，$O(n^4)$ の計算時間が必要になる．

そこで X_j を展開して，

$$
\mathrm{L}_i \quad \rightarrow \quad \mathrm{X}_j \; \mathrm{L}_i \quad \text{where} \quad j \overset{\frown}{} i
$$

であったところを

$$
\mathrm{L}_i \quad \rightarrow \quad \mathrm{L}_j \; \underline{{}_j\mathrm{R} \; \mathrm{L}_i} \quad \text{where} \quad j \overset{\frown}{} i
$$

と変換する．ここで下線部を置き換える新たな非終端記号 ${}_i\mathrm{M}_j$ を導入し，以下のように規則を変更する．

$$
\begin{array}{lcll}
{}_i\mathrm{M}_j & \rightarrow & {}_i\mathrm{R} \; \mathrm{L}_j & \\
\mathrm{L}_i & \rightarrow & \mathrm{L}_j \; {}_j\mathrm{M}_i & \text{where} \quad j \overset{\frown}{} i
\end{array}
$$

${}_i\mathrm{R}$ に関しても同様にして，X_i の出現をなくす．最終的に以下の規則を得る．

$$
\begin{array}{lcll}
\mathrm{S} & \rightarrow & \mathrm{L}_i \; {}_i\mathrm{R} & \text{where} \quad \overset{\mathrm{ROOT}}{\frown} i \\
\mathrm{L}_i & \rightarrow & i_l & \\
\mathrm{L}_i & \rightarrow & \mathrm{L}_j \; {}_j\mathrm{M}_i & \text{where} \quad j \overset{\frown}{} i \\
{}_i\mathrm{R} & \rightarrow & i_r & \\
{}_i\mathrm{R} & \rightarrow & {}_i\mathrm{M}_j \; {}_j\mathrm{R} & \text{where} \quad i \overset{\frown}{} j \\
{}_i\mathrm{M}_j & \rightarrow & {}_i\mathrm{R} \; \mathrm{L}_j &
\end{array}
$$

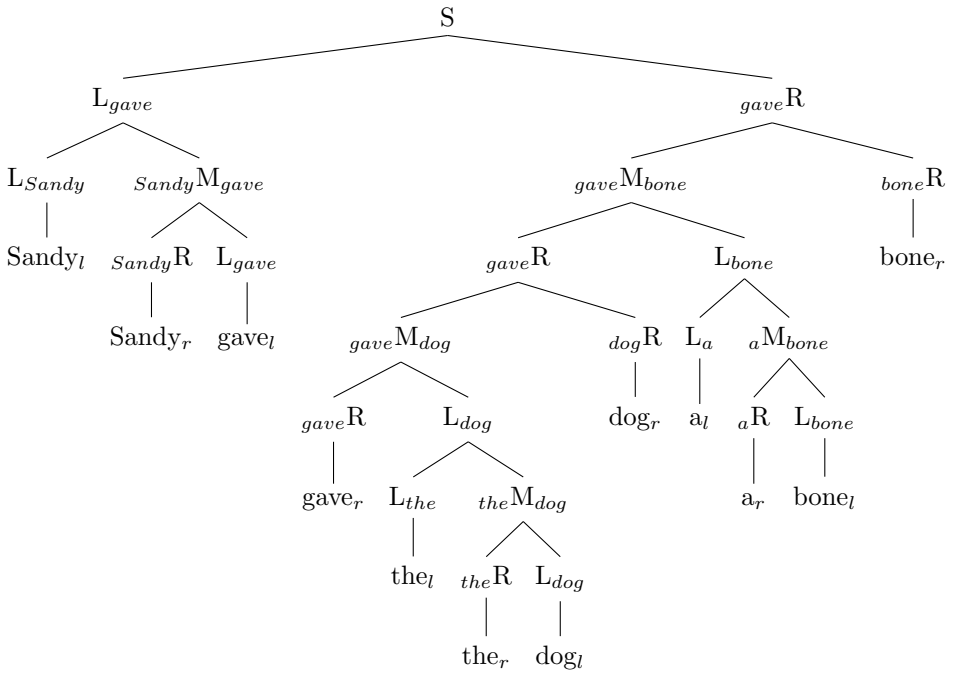


図 2.6: 図 2.1 の依存構造に対して $O(n^3)$ 版の split-head の CFG を用いたときの導出

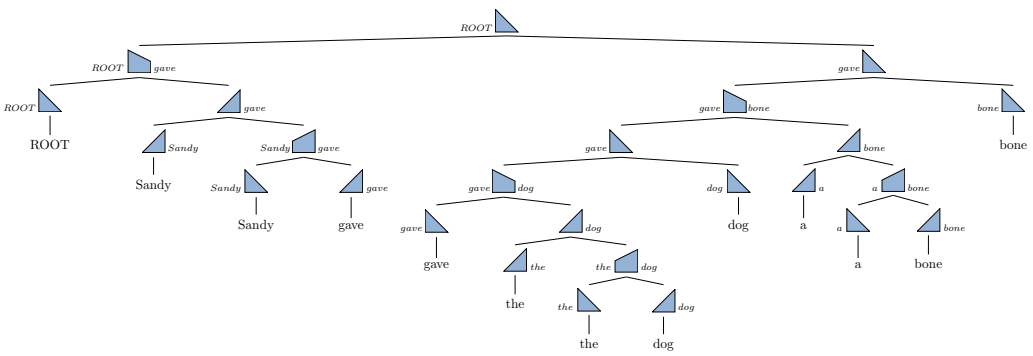


図 2.7: 図 2.1 の依存構造に対する Eisner アルゴリズムの導出

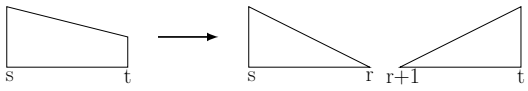
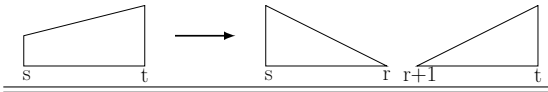
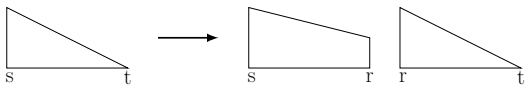
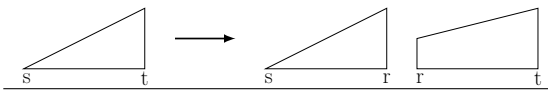
未完成の Span を作る規則	対応する記号表記
	${}_sMR_t \rightarrow {}_sR\ L_t$
	${}_sML_t \rightarrow {}_sR\ L_t$
完成した Span を作る規則	**対応する記号表記**
	${}_sR \rightarrow {}_sMR_r\ {}_rR$
	$L_t \rightarrow L_r\ {}_rML_t$

表 2.3: 表 2.2 に対応する記号表記

このようにすることで，計算時間を $O(n^3)$ にすることができる．この場合の導出は図 2.6 のようになる．Johnson によって提案されたこの文法は，Eisner アルゴリズムに近く，CKY アルゴリズムを用いる際には本質的に Eisner アルゴリズムと等価である．主な違いは，Eisner アルゴリズムで二つの向きの台形で表された非終端記号が一つの非終端記号 M に対応することである．Eisner アルゴリズムによる導出木は図 2.7 のようになる．

Eisner アルゴリズムで図形によって表記されていた Span を Johnson の文法表記のような記号を用いて表すこともできる．Johnson の文法で M として表記されていた非終端記号に対応する二つの台形を，左向きと右向きでそれぞれ ML, MR として表記し，三角形を L と R で表すと表 2.3 のように対応する記号表記で表すことができる．本論文では規則の表記に図形による表記と文字記号による表記の両方を用いる．

2.2 並列構造のアノテーション

英語においては依存構造がそのままアノテーションされた大規模コーパスがなく，Penn Treebank を変換することで依存構造グラフを得て，これを利用して依存構造の解析器が開発されている．句構造木から依存構造グラフへの変換は，各句構造で Head を選択する Head Percolation Rule を定義し，句構造の Head 以外の要

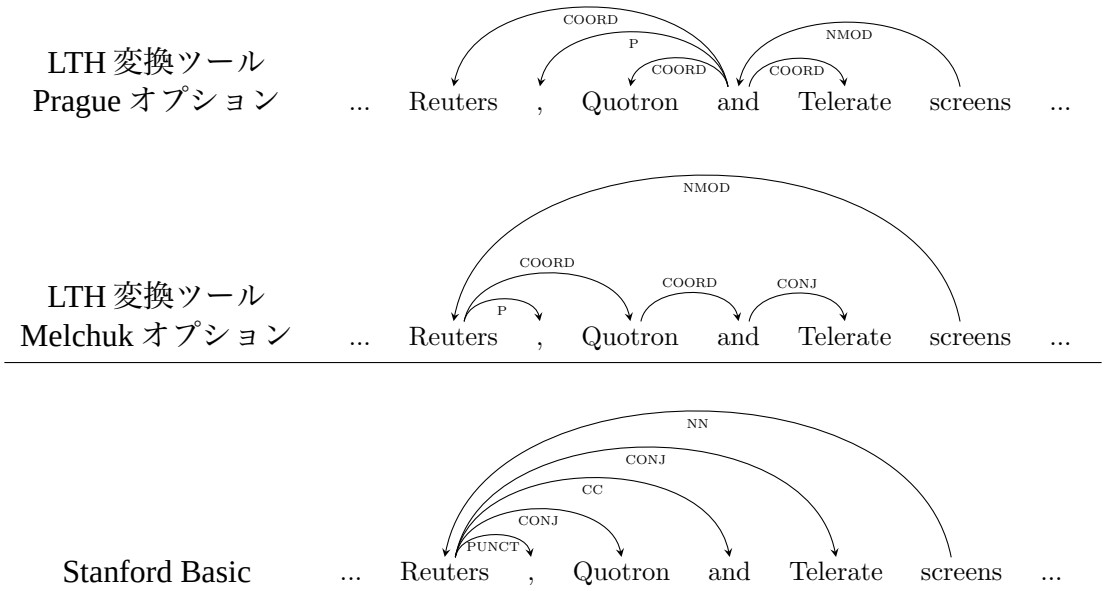


表 2.4: 並列構造の各種変換

素が Head に係るように変換することによって実現される．依存構造グラフへの変換は一通りに限られるものではなく，複数の Head Percolation Rule が提案されている．それぞれの規則で，助詞や関係節などにおける依存関係の方向が異なっているほか，並列構造における依存構造グラフの仕様にも違いがある．

表 2.4 は [Johansson and Nugues, 2007] による LTH 変換ツール [1] を用いた各種依存構造への変換結果と Stanford parser によって変換された Stanford アノテーション [De Marneffe and Manning, 2008] への変換結果である．LTH 変換ツールは出力するアノテーションの種類を与えるオプションによって切り替えることができる．Prague オプションを与えた場合の変換では，並列される項目が “ and ” のような並列マーカの左側と右側に現れ，依存関係の方向に一貫性がない．Melchk オプションを与えた場合の変換では，“ , ” がその左側の単語の Dependent になり，右側の単語も左側の単語の Dependent となる．一方で “ and ” のような並列マーカが現れると，並列マーカは左側の項目の Dependent になると同時に右側の項目の Head になる．この方法だと，“ , ” が組み合わせる部分木と並列マーカが組み合わせる部分木に一貫性がなく，その分だけ解析時の規則が複雑になる場合があると考えられる．

Stanford アノテーションには複数の種類があるが，本研究では木構造となる Basic 変換を扱う．以降，この変換を Stanford Basic アノテーションと呼ぶ．この変換は，

[1] http://nlp.cs.lth.se/software/treebank_converter/

並列範囲の最初の単語を Head とし，並列範囲内の他の単語が全てその子孫になる．並列範囲の最初の項目と並列関係にある項目の Head へのエッジの向きは全て右向きに統一される．並列範囲内部の依存関係が範囲を示すための関係にとどまるため，並列されている項目それぞれの範囲に関する曖昧性が残る可能性が考えられるが，並列構造の解析のしやすさの観点では依存方向が統一されているために規則の複雑化を避けられる利点がある．

2.3 並列構造解析

並列構造の曖昧性は並列される項目の範囲の曖昧性であり，構文構造の曖昧性に含まれる．しかし構文解析が扱う問題の中でも特に難しく，並列構造だけに着目して解析を試みるアプローチが数多く提案されてきた．並列構造の曖昧性を構文解析の問題の一部として，他の構文的曖昧性と同時に扱ってこれを直接解消する手法は未だ提案されていない．しかし，並列構造解析と従来の構文解析の結果を両方利用して構文解析の精度が向上することが確認されている．

2.3.1 名詞句の並列構造解析

並列構造の曖昧性に着目した解析手法には，その中でもさらに名詞句内部の並列構造に限定した研究がある．[Resnik, 1999] は “n_1 and n_2 n_3”（n_i は名詞）という形式の単語列に関する並列構造の曖昧性解消を行った．この並列構造は，“(n_1 and n_2 n_3)” あるいは “(n_1 and n_2) n_3” と解釈できる．Resnik は，n_1 と n_2 および n_1 と n_3 に関してコーパスから計算した意味的な類似度や，それらの単複の一致が，曖昧性解消のために重要な情報であると報告している．[Nakov and Hearst, 2005] はウェブ検索による単語の共起頻度を，[Chantree et al., 2005] はコーパス上の文脈（分布）類似度を，それぞれ類似したタスクに利用している．

2.3.2 文全体を対象にした並列構造解析

文全体を解析対象とする英語の並列構造解析の研究には，[Agarwal and Boggess, 1992]，[Hogan, 2007]，[Okumura and Muraki, 1994]，[Buyko et al., 2007]，[Buyko and Hahn, 2008]，[Hara and Shimbo, 2007]，[Hara et al., 2009] がある．

特定のパターンで現れる並列構造に限定した手法

[Agarwal and Boggess, 1992] は発見的に作成したパターンを用い, 文に現れる等位接続詞に対し並列項目対を文から抽出する手法を提案した．[Okumura and Muraki, 1994] は発見的に作成したパターンを用いて並列項目対となる候補の集合を特定した後，形態素解析，意味解析などを用いて獲得した素性をもとに算出する類似度スコアが最大となる並列項目対を候補集合から選ぶ手法を提案した．[Hogan, 2007] は，名詞並列構造が含まれることがわかっている文を対象に，文の構造解析を行った．並列項目間の対称性と，並列関係にある主辞単語間の依存関係を取り込んだ生成モデルを提案し，このモデルを Bikel parser[2] [Bikel, 2002] が出力する *n*-best 解のリランキングに用いた．

機械学習に基づき，パターンを極力限定しない手法

[Buyko et al., 2007]， [Buyko and Hahn, 2008]，[Hara and Shimbo, 2007]，[Hara et al., 2009] は並列構造解析に識別学習法を適用した．Buyko らは linear-chain CRF を用い，Hara らは系列アライメントに基づく並列項目間の編集距離モデルをパーセプトロンを用いて学習した．

系列アライメントに基づく手法

系列アライメントによって並列項目の類似度を捉える手法は [Kurohashi and Nagao, 1994] による日本語の並列構造解析法で最初に提案された．Kurohashi らによる日本語の並列構造解析は，日本語係り受け解析器 KNP[3] [黒橋禎夫, 1998]，[黒橋禎夫, 2000] の処理の一部として組み込まれている．KNP ではまず並列キーが入力文に含まれているか調べる．含まれている場合には，並列キーの文節前後のあらゆる文節列の対に対し，どの対が並列する 2 つの文節列であるかを決定するためのスコアを計算する．そしてスコア最大となる文節列の対を並列項目の範囲として出力する．ただし，スコアが基準値以下の場合には並列句は存在しないと判定する．Hara らの手法も並列キーの文節前後における並列項目対の候補に対し，最良の候補を選び出すためのスコアを計算するが，計算手法や素性を改良し，機械学習が適用できる手法に発展させた．

[Hara and Shimbo, 2007] による英語の並列構造解析は，文全体をアライメントグラフあるいは編集グラフ [Gusfield, 1997] と呼ばれるグラフにあてはめる．このグ

[2] http://web.mit.edu/6.863/tools/dbparser/

[3] http://nlp.ist.i.kyoto-u.ac.jp/index.php?KNP

ラフは遺伝子の系列アライメントに用いられているもので，水平，垂直，対角の三方向の辺が存在する．対角方向の辺は，対応する行方向の文字を列方向の文字に置換する編集操作を表す．水平方向の辺は列方向の文字を挿入する編集操作に，垂直方向の辺は行方向の文字を削除する編集操作にそれぞれ対応する．これらの編集操作の適用と，その時の周辺にある単語や品詞情報をスコア計算の素性として利用している．

Hara らの用いた手法を除くほとんどの既存研究は，限られた形の並列構造を，一見して十分すぎる情報を与えられた状況で解析するという問題設定となっている．一方で Hara らの手法は並列構造解析に解析対象の単語列と品詞情報のみを用いている．

2.3.3 並列構造解析と従来の構文解析器を利用した総合的な解析

[Hanamoto et al., 2012] は並列構造の曖昧性解消に双対分解を用いる統計的解析モデルを提案した．この解析モデルは文中の並列構造部分における解析に Hara らのアライメントベースの解析法を用い，構文木全体の解析には HPSG [Pollard, 1994] による構文解析器 Enju [Miyao and Tsujii, 2004] を用いている．双対分解を用いてそれぞれの解析結果を統合する手法になっているが，構文解析の一部として並列構造解析を取り入れた方法は提案されていない．

2.3.4 並列構造解析で有効な素性

並列構造解析には，並列される項目間の表層形の類似度と意味の類似度が有効であることが [Kurohashi and Nagao, 1992] によって示されている．

例えば，“ advanced computer games and TV entertainment systems ” という文について，どの部分が並列項目の範囲となっているかを考えると，図 2.8 に示すように，複数の範囲の候補が挙げられる．ここで，図 2.9 に示すように “ and ” の左側と右側で表層形に注目してみると，“ games ” と “ systems ” が複数形という点で共通している．また，“ computer games ” と “ TV entertainment systems ” は同様に名詞の並びとなっており，意味も類似していると考えられる．

並列構造に存在する，並列項目間の構造的あるいは意味的な類似度の特徴を捉えることを目的として各種の解析法が提案されてきたといえる．しかし [Hara et al., 2009] では，類似度を利用した並列構造解析は，並列項目の種類によって精度が大きく変わることが示された．その分析を表 2.5 に一部抜粋する．

例えば，この表の NP で表される名詞句の並列構造は，表層や品詞の素性を用いることで通常の構文解析器より高精度に範囲同定が可能である一方で，表の VP で

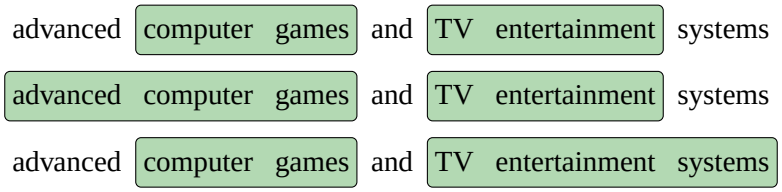


図 2.8: 複数考えられる並列項目の範囲

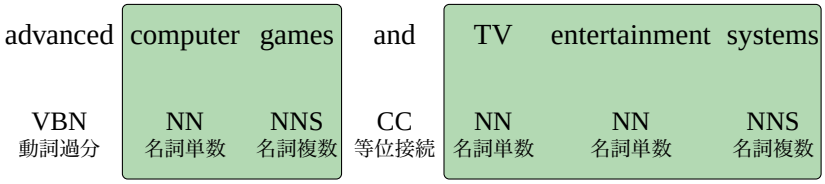


図 2.9: 表層と意味の類似性

COOD	#	Hara method	Bikel parser
Overall	3598	**61.5**	52.1
NP	2317	**64.2**	45.5
VP	465	54.2	**67.7**
ADJP	321	**80.4**	66.4
S	188	22.9	**67.0**
PP	167	**59.9**	53.3
UCP	60	**36.7**	18.3
SBAR	56	51.8	**85.7**
ADVP	21	85.7	**90.5**
Others	3	**66.7**	33.3

表 2.5: [Hara et al., 2009] による並列項目の種類別の精度分析（一部）

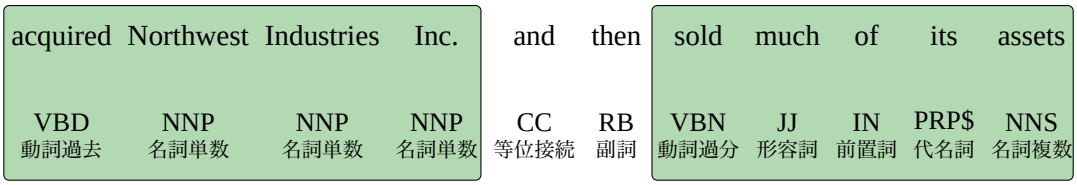


図 2.10: 表層と意味的類似性だけでは捉えられない動詞並列の例

表される動詞句や，S で表される文並列などでは，同様の手法では精度良く捉えることができないことが示されている．

実際，文並列や動詞並列では図 2.10 に示すように，表層も単語の意味もその系列もほとんど似ていない場合が存在する．この場合は，単純に類似度をはかるだけでは並列項目の範囲を同定することは難しい．このような範囲の同定については，通常の構文解析器の方が精度が高い傾向がある．

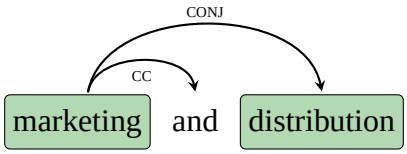
これらのことから，並列構造の範囲同定で表層や意味の類似度を素性として使うべきかは，並列される項目候補の種類によって判断されるべきであると考えられる．よって，並列構造解析を独立に行うのではなく，これを構文解析と同時に行うことで，並列項目の候補ごとに適切な素性を使うことが並列構造解析にとって必要だと考えられる．そのため，本研究では構文解析と同時に並列構造解析を行うことを目的としている．

第3章　並列構造を扱うための規則

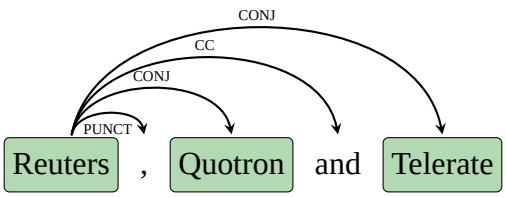
本研究では Eisner アルゴリズムによる依存構造解析の過程で，並列関係にある句の範囲に相当する Span を捉える規則を提案する．一般的に Eisner 規則では，並列構造を優先的に組み上げることができない．例えば Stanford アノテーションによる並列構造の導出は図 3.4 のようになり，個々の規則適用時に並列項目らしさをそのまま素性として利用することはできない．そこで，図 3.5 のように導出することで，並列構造において各規則の適用時に並列項目らしさを素性として利用できるような規則を提案する．

3.1　対象とする並列構造

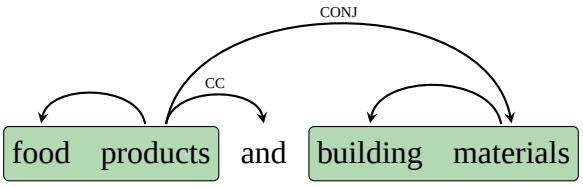
英語の文において，並列構造は同じようなパターンを持っていると考えられる．多くの場合，並列される項目は “ , ” などの限られた種類の文字で区切られ，最後の並列項目の前には “ and ” や “ or ” のような並列マーカが出現する．並列される項目が 2 つの場合は並列マーカだけで区切られ，“ A and B ” の形となり，Stanford Basic アノテーションでは以下のような形の依存構造となる．



並列される項目が 3 つ以上の場合は “ A , B and C ” や “ A , B , C and D ” といった形となり，Stanford Basic アノテーションでは以下のような形の依存構造となる．



Stanford Basic アノテーションでは並列構造の先頭の項目が並列構造の Head になり，並列構造に関わる PUNCT，CC，CONJ は全て先頭の項目の子となる．並列項目の中に通常の依存関係がある場合は，その依存関係は通常通りの形に保たれ，以下のような形となる．



本研究では，これらの並列構造に共通する，先頭の並列項目を並列構造の Head とし，その並列構造に関わる依存関係は全てそこからの右側の子になるという形を扱うための規則を考え，その形を扱う際に並列項目の類似度を調べられるようにすることを考える．並列構造の形についてのこの仮定でどの程度の並列構造が網羅できているかについては 4 章で検証する．

3.2 提案規則

並列構造のアノテーションが Stanford アノテーションである場合に Eisner アルゴリズムによってこれを捉えるための提案規則を表 3.1 と表 3.2 に示す．

Eisner アルゴリズムで使われていた規則では，左向きの三角形と右向きの三角形を作り，それらを組み合わせるときには向き合った三角形にすることで各三角形の Head から Head への依存関係を作っていた．本論文では，これに加えて，図 3.1 に示すような，両側に向いた三角形を作ることを考える．この場合，2.1.2 節で述べた擬似曖昧性の問題が生じると考えられるが，これを解消する方法については 3.3 節で後述する．この図は，図の右側にある反対に向いた 2 つの三角形を組み合わせて，図の左側のような 1 つの三角形を作ることを意味している．この三角形においても，図の j にあたる単語が Head となる．以降この三角形を conjunct と呼ぶ．conjunct では，Head の位置が Span の内側のどの単語の位置もとることができるようになるため，これまで Span に対応する単語列の範囲を示していた i と j の他に，Head の位置を保持しておく必要がある．これは図の左側にある conjunct 内の j に対応し，j は conjunct ごとに保持しておく必要があることを示している．

ここで，conjunct の作り方をもう一つ考える．図 3.2 に示す規則は 2 つの隣り合う conjunct を組み合わせて 1 つの conjunct を作ることを意味する．この規則ではそれぞれの conjunct の Head から Head への依存関係を作らなければ，それぞれの

PUNCTMARK	→	L, ,R
CCMARK	→	L& &R
BEFORECCCONJ	→	(PUNCTMARK) L R (CCMARK)
BEFORECCCONJ	→	COORDCOMPLETE
LASTCONJ	→	(CCMARK) L R
LASTCONJ	→	COORDCOMPLETE
MIDDLECONJ	→	L R (PUNCTMARK/CCMARK)
MIDDLECONJ	→	INCOMPLETECONJ
MIDDLECONJ	→	COORDCOMPLETE
MIDDLECONJ	→	INCOMPLETECONJ BEFORECCCONJ
INCOMPLETECONJ	→	MIDDLECONJ PUNCTMARK
NEXTLAST	→	MIDDLECONJ CCMARK
COORDCOMPLETE	→	NEXTLAST LASTCONJ
COORDCOMPLETE	→	RIGHTINCOMPLETE R
RIGHTINCOMPLETE	→	COORDCOMPLETE L
L	→	COORDCOMPLETE L
R	→	R COORDCOMPLETE

表 3.1: Stanford アノテーションのための並列構造規則（文字表記）

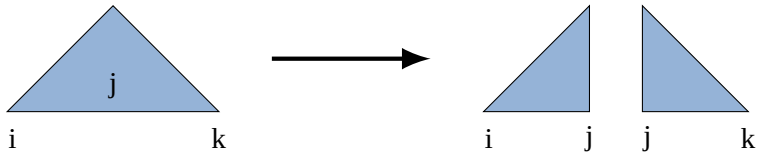


図 3.1: 提案規則：conjunct の作成

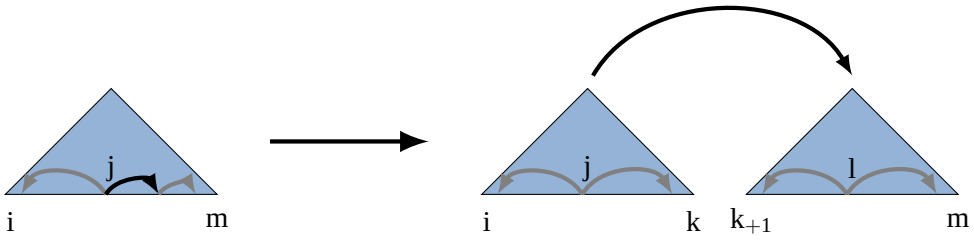


図 3.2: 提案規則：conjunct の組み合わせ

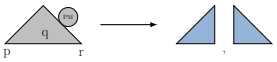
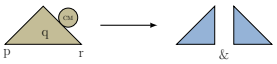
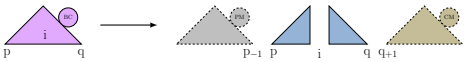
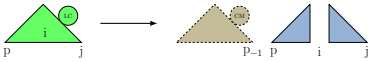
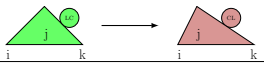
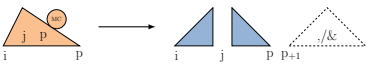
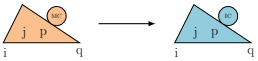
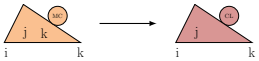
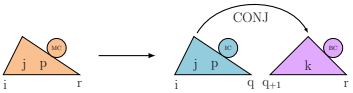
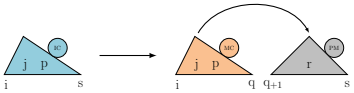
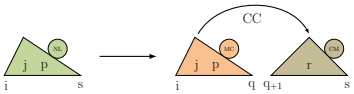
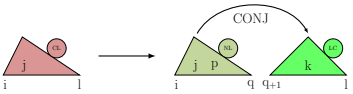
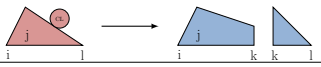
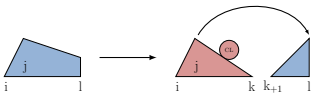
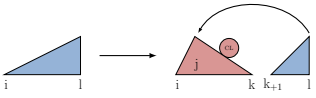
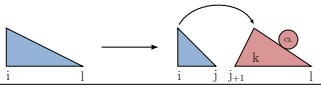
PUNCTMARK		
CCMARK		
BEFORECCCONJ		
BEFORECCCONJ		
LASTCONJ		
LASTCONJ		
MIDDLECONJ		
MIDDLECONJ		
MIDDLECONJ		
MIDDLECONJ		係り受け素性 j → k (CONJ) 並列素性 (i-p)(q+1-r)
INCOMPLETECONJ		係り受け素性 j → r
NEXTLAST		係り受け素性 j → r (CC)
COORDCOMPLETE		係り受け素性 j → k (CONJ) 並列素性 (i-p)(q+1-l)
COORDCOMPLETE		
RIGHTINCOMPLETE		係り受け素性 j → l
LEFTCOMPLETE		係り受け素性 l → j
RIGHTCOMPLETE		係り受け素性 i → k

表 3.2: Stanford アノテーションのための並列構造規則（図形表記）

conjunct に含まれる木が接続されないため，この依存関係のスコアを計算し，図の左側にある conjunct にそのスコアが付与される．

並列項目を接続する並列マーカである “ , ” や “ & ” はそれぞれ，特別な conjunct に変換される．“ , ” は並列項目の列挙が後にも複数続くことを意味する conjunct になり，“ & ” は並列項目の列挙がその語の次に現れる項目で終わることを意味する conjunct になる．“ , ” や “ & ” 以外にも，類似の役割を持つ語がある．例えば，“ , ” に近い役割の語には “ ; ” や “ / ” があり，“ & ” と同じか，あるいは近い役割の語には “ and ”,“ or ”,“ but ”,“ as well as ” などがある．本論文では，この中で特に使用頻度の高い “ , ”,“ & ” に絞って規則を表現するが，それぞれ並列項目がその後に続く場合の区切りになる表現，並列項目の最後の区切りになる表現として扱うため，“ & ” に適用される規則には “ as well as ” や “ , and ” などを加えることで，より一般的な並列表現に対応できる．

- CC は並列マーカに現れ，その Head が並列範囲の最初の並列項目の Head になる．
- CCMARK は並列範囲内を並列項目に区切る記号のうち，並列範囲の終わりを探させるマークとして作成する．
- PUNCTMARK も並列範囲内を並列項目に区切る記号であるが，並列範囲がまだ続くとして次の並列項目を探させるマークとして作成する．
- MIDDLECONJ は句としては完成できる Span を意味するが，並列マーカをまだ取り込んでいないため，並列構造としては完成していないことを意味する．
- 単語や句単体でも MIDDLECONJ になることができ，その場合は並列構造の最初の並列項目であることを意味する．最初の MIDDLECONJ が作られるときにその Span の終わりを p とし，各状態において並列範囲の終わりを示す p は最初の並列項目の範囲の終わりを引き継いでいく．後に並列項目候補となる Span と組み合わせるときに，引き継がれる最初の並列項目の範囲を用いて，同じ並列構造の一部としての並列項目の範囲となるか計算することができるようになる．
- INCOMPLETECONJ は MIDDLECONJ 句として完成していない Span を意味し，続く Span は並列構造の CONJ になっていなければならない．
- NEXTLAST も句として完成していない Span を意味し，続く単語列は並列構造の最後の CONJ になっていなければならない．

- LASTCONJ は並列マーカより後に現れる Span を意味する．
- COMPLETECONJ は NEXTLAST と LASTCONJ の組み合わせからなり，並列構造が完成したことを意味する．
- BEFORECCCONJ は並列マーカ手前の並列項目の Span を意味し，手前の Span は PUNCTMARK，続く Span は CCMARK でなければならない制約がある．

提案した規則は元々の Eisner 規則と足しあわせることで並列構造以外も含んだ構文木全体を組み上げることを考えることができる．しかし新たに提案した並列構造のための規則によって組み上げられる構造は，元々の Eisner 規則によっても組み上げることができるため，提案規則を元々の Eisner 規則とそのまま足しあわせると組み上げられる構造が一意に決まらなくなる．組み上げられる構造が一意に決まらなくなると，並列構造の特殊な依存構造が通常の Eisner 規則の方でも学習されてしまう．そこで，並列構造は提案した規則でしか組み上げられないようにする必要がある．そのために，元々の Eisner 規則では品詞が CC であるような単語を子に持つ Span を作れなくすることを考える．こうすると，並列構造の内側の構文木は元々の Eisner 規則からは導出されなくなり，導出される木が一意に定まる．

2 つの conjunct 同士を組み合わせるとき，常に右方向への依存関係が生じるものとし，赤の三角形と青の三角形を組み合わせるとき，常に青から赤への依存関係が生じるものとする．これらはいずれも，組み合わせる三角形の Head から Head への依存関係となる．

同様に単語の位置を示す変数 (p, q, r, s) は並列構造の区切りとなる単語の位置によって束縛された変数を示し，(i, j, k, l) は自由変数を示す．

破線で示された三角形は，その種類の三角形が存在する場合にしか規則を適用できないという制約を示す．

三角形の外側に示された変数は従来のアルゴリズムと同様に Span を決定する変数を表し，三角形の内側に示された変数は Span に保持しておく変数を表す．表 3.2 の中央にある規則のうち COORDCOMPLETE 以外は，並列される範囲となる他の句を比較するために最初の並列項目の範囲を変数 p として保持している．最初の並列項目の範囲は Span の左端，すなわち i からこの変数 p までの範囲として捉える．これを，BEFORECCCONJ, LASTCONJ を組み合わせるときにその CONJ 範囲と比較し，並列項目対らしさを計算するのに利用できる．

通常の依存関係を捉えるための規則にこれを加えることで，並列構造の依存関係と通常の依存関係を共に扱うことができ，並列構造の依存関係に関しては，並

列される Span の類似度を測ることができれば，並列構造の確からしさを依存構造の部分木のスコアとして組み合わせることができる．

元の Eisner 規則で図 2.3 のように Span を広げていた解析過程は，提案規則規則では図 3.3 のようになり，並列項目の候補が一つの Span に収まった状態で正解の構文木を作ることができる．

また，元の Eisner 規則で図 3.4 のように導出されていたものは，提案規則では図 3.5 に示す導出となる．

事例として数は少ないが，並列構造全体がその並列構造の外にある単語に対する Head になる場合がある．そのような文は，例えば図 3.6 のようになっており，この場合の導出を図 3.7 に示す．

また，並列構造には並列項目が全て同列に並列されるものの他に，並列構造の並列項目の中に並列構造が入れ子として含まれている場合がある．このような例には図 3.8 があり，この場合の導出は図 3.9 のようになる．

3.3 提案規則における擬似曖昧性の回避

Stanford アノテーションでは並列構造において並列範囲の先頭の項目が並列関係の Head となり，それと並列される他の並列項目が全て右方向の Dependent となるため，この並列関係を表す部分木は必ず左下がりの木構造となる．この並列関係の部分木に関しては，部分木の向きの曖昧性が生じないため，Head を Span の端に置かない場合でも擬似曖昧性は生じない．

そこで，Head が Span の内側にあるような規則の使用を並列関係を組み上げる規則に限定することで，並列関係を構成する部分木に擬似曖昧性がないことから，構文木全体として擬似曖昧性を持たない構文解析ができる．

3.4 提案規則の計算量

本来，Head が Span の内側に来ることを許容すると，Span の位置の他に Head の位置を考慮するために計算量は $O(n^5)$ になる．しかし，提案規則では Head の位置を別に考慮するのは並列関係を構成する規則に限定し，さらに，並列関係を構成する規則の分割位置は並列項目の区切り文字に使われる文字の位置に制限される．これにより，時間計算量を支配する単語位置のうち一つは区切り文字の数に支配されることになり，$O(n^4m)$（n は単語の数，m は区切り文字の数）の計算量となり，多くの場合で m は n より十分小さいため，$O(n^4)$ とほぼ同等の計算量になると考えられる．

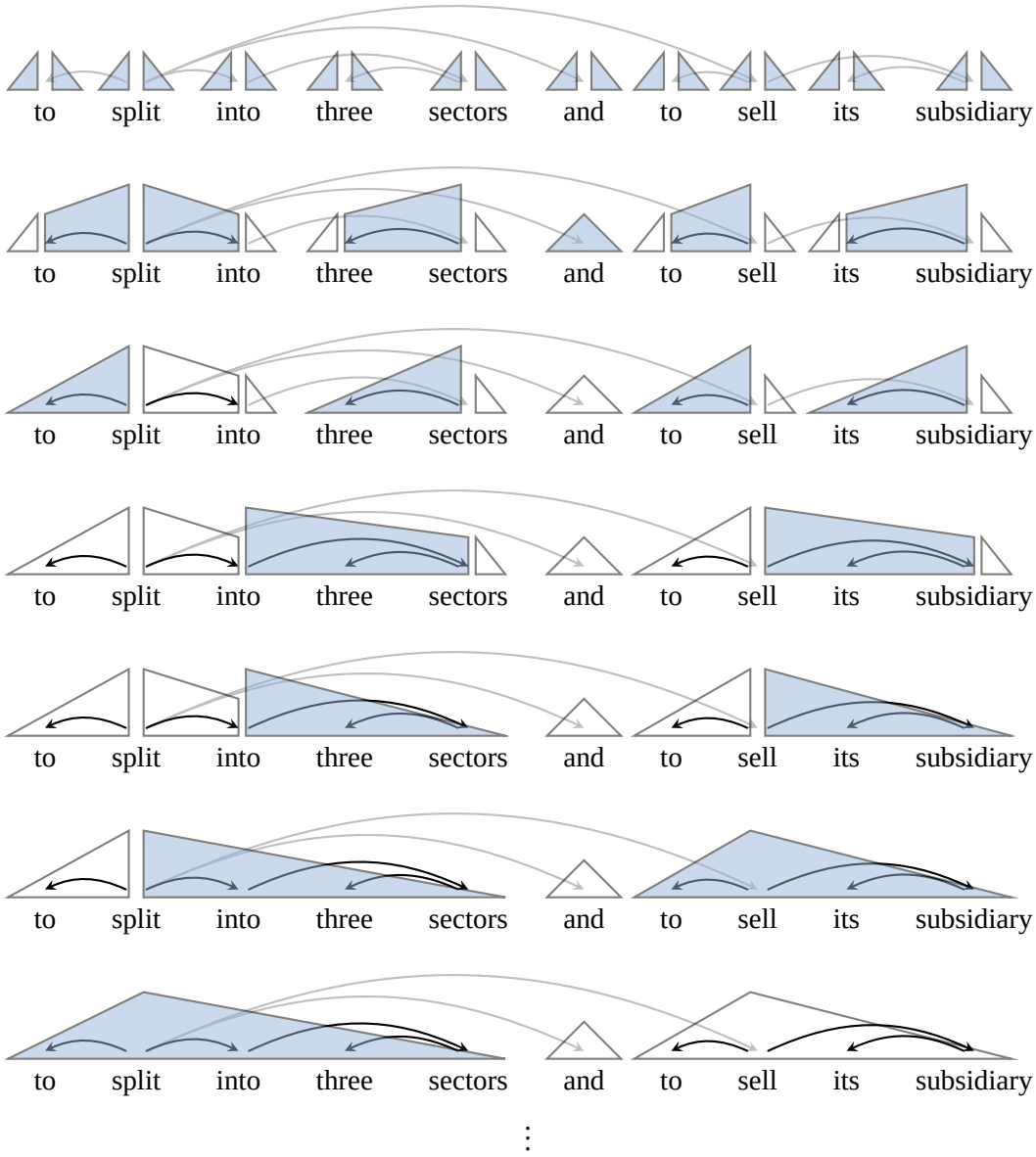


図 3.3: 提案規則による解析過程

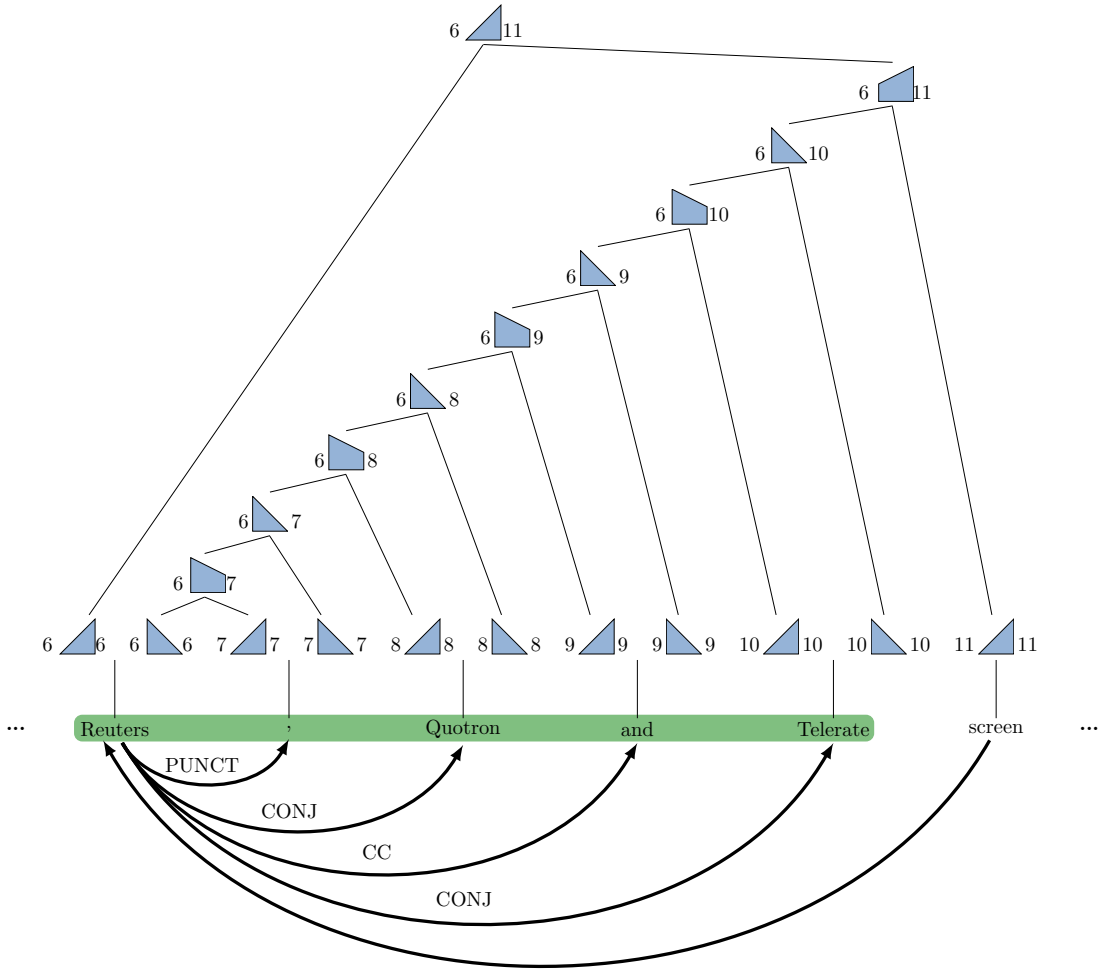


図 3.4: Eisner 規則 (1st order) による導出例

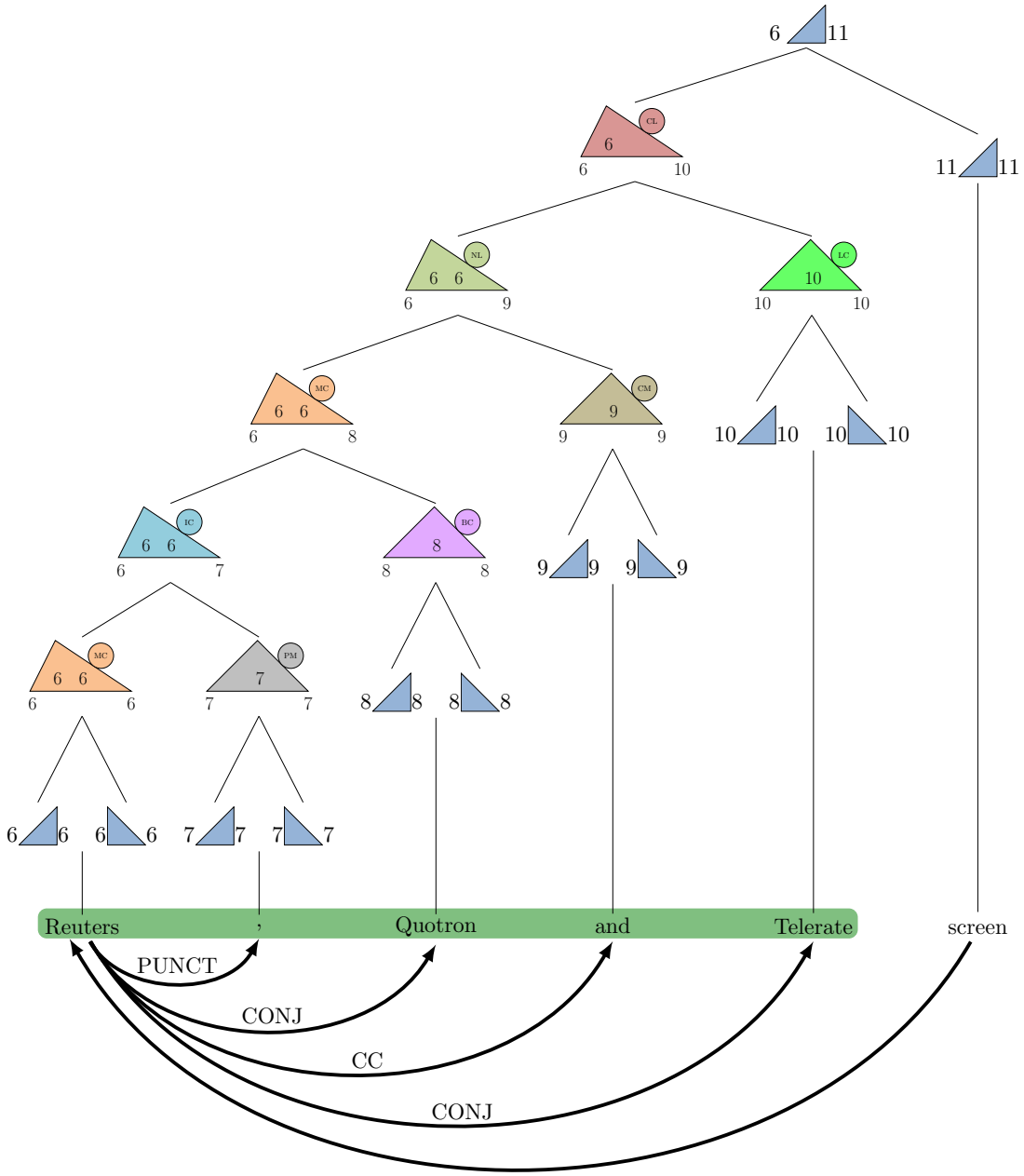


図 3.5: 提案規則による並列構造の導出例

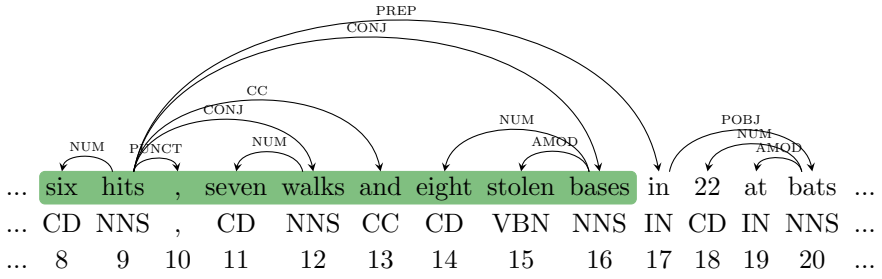


図 3.6: 並列構造が他の単語の Head になる例

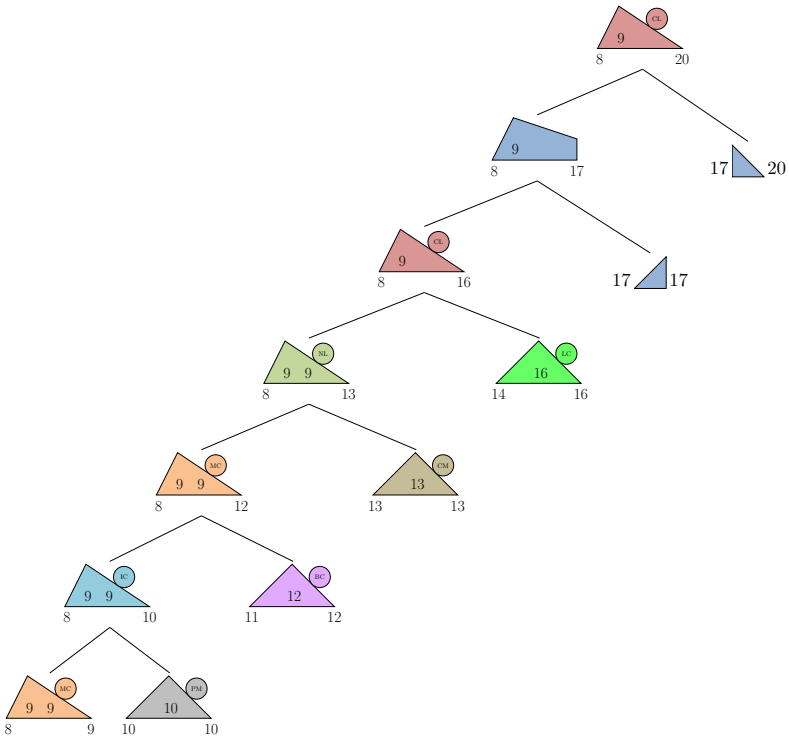


図 3.7: 図 3.6 における並列構造の導出例

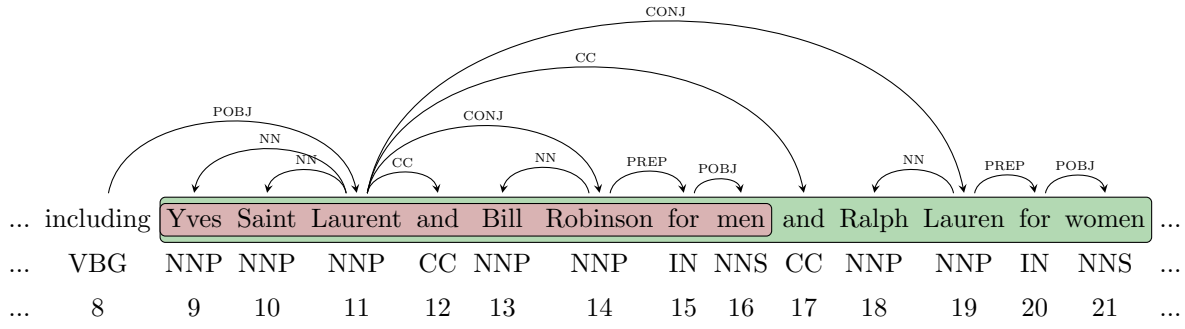


図 3.8: 並列構造が入れ子になる例

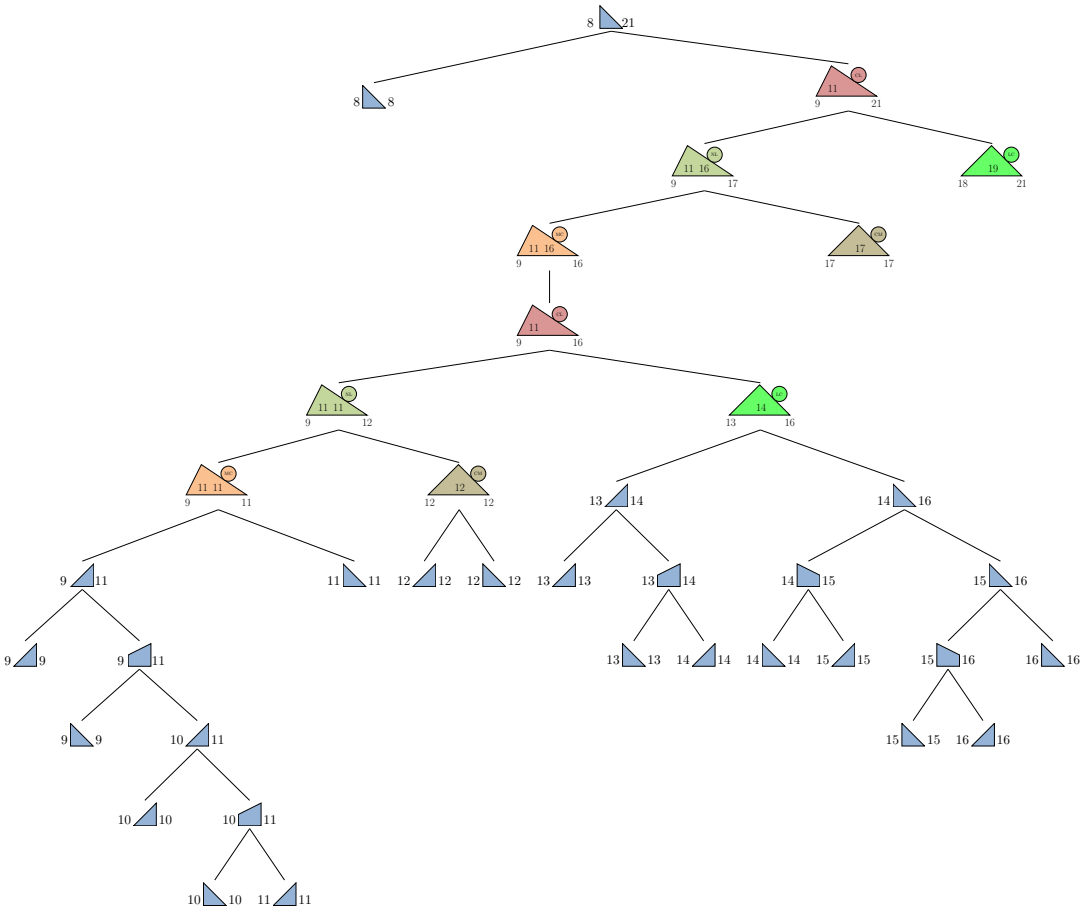


図 3.9: 図 3.8 における並列構造の導出例

第4章　提案規則による正解構文木の作成

4.1　依存構造アノテーションから正解構文木へ変換するアルゴリズム

Eisner 法など，機械学習による依存構造解析では，正解の構文木を作成して学習し，新たな入力に対する構文木を予測する．構文木を解析する手がかりになるのは，正解の構文木における依存構造のラベルと，依存構造があるときに存在した単語や品詞，距離などの素性である．

以下に，Stanford アノテーションによる依存構造木から提案規則を用いる正解の構文木を自動的に作成するアルゴリズムを説明する．

本章ではアルゴリズム中の非終端記号を 3 章で提案した非終端記号に対応する略記にて表す．この表記の対応については表 4.1 に示す．

Convert

与えられた依存構造木を導出木に変換するためには, 関数 Convert を呼ぶ. 得られた導出木は S 式 (文字列) として返される. 全ての導出規則は 2 つの非終端記号を組み合わせて新しい非終端記号を導く. これらをそれぞれ A_1, A_2, C とすると, この導出は（`C A1 A2`）と書かれる. A_1, A_2 もまた同様にそれぞれ 2 つの非終端記号の組み合わせからなり，終端記号に達するまで再帰的に導出される．

Convert は 3 ステップからなる．

1. 最初のステップは, MarkRoles 関数を呼び出し, 依存木の各辺 (個々の依存関係) に対して, 並列構造に関係するかどうか・どんな役割か, を判定し, マーク付けを行う. 結果は role という配列 (辺の子接点からマークへの写像を表す) に返される．

2. 各節点に leftS, rightS という 2 種類のスロットを割当てる. これらスロットには, 該当節点の左側の子孫の導出木を表す S 式 (leftS), 右側の子孫の導出木を

略記	提案規則の非終端記号
PM	PUNCTMARK
CM	CCMARK
BC	BEFORECCCONJ
LC	LASTCONJ
MC	MIDDLECONJ
IC	INCOMPLETECONJ
NL	NEXTLAST
CL	COORDCOMPLETE
RI	RIGHTINCOMPLETE

表 4.1: 提案規則の非終端記号と対応する略記

表す S 式 (rightS), を表す. これらスロットは, 次のステップで, 計算が行われるが, その前に, ここで単語位置と左右を表す文字列 (L および R と単語位置を組み合わせたアトム (文字列) で初期化される. これは Eisner アルゴリズムにおける, 単語の左右の三角形に相当する.

3. 手続き Traverse を root 節点を引数として呼び出す. ステップ 1 で作った, role 配列も変換の際に使うため, これも第 2 引数として渡す. この手続きは, leftS, rightS に格納された S 式を依存木の最下部から順に更新することで, 最終的に全体の導出木を得る.

MarkRoles

関数 MarkRoles は, 依存木の各節点 v を訪問し, その右側の子節点のうち,

- r = Type が CONJ のうち最も右 (文末に近い位置) にあるもの
- ℓ = Type が CONJ または CC のもののうち, 最左 (文頭に近い位置) のもの

をまず見つける.

- v の子節点 c のうち, type が conj, cc であるものについては, $c \leq r$ の場合, それぞれ role も conj, cc であるとする. $r < c$ の場合 (r の定義により, これは cc についてしか起こりえない) は依存構造のアノテーション間違いの可能性がある.

- type が punct かつ form が ‘,’ (カンマ) または ‘; (セミコロン)’ の場合には, $\ell < c < r$ の場合, role = punct とする. また, 親節点 v の子供の内, ℓ の直前のものが type = punct, form . がカンマまたはセミコロンの場合にも, role = punct とする.

Traverse

手続き Traverse は, 引数として与えられた節点の

- 左側の子孫を S 式で表現された導出木に変換する手続き ReduceLeft
- 右側の子孫を S 式の導出木に変換する手続き ReduceRight

の二つの関数を順に呼び出す. 主手続きの Convert からだけでなく, ReduceLeft, ReduceRight 内から再帰的に呼ばれる.

ReduceLeft

引数として与えられた節点 v の左側の子供を, v に近いものから順に, 手続き Traverse を呼ぶ. 子供 c の leftS, rightS として得られた結果を, v の導出ステップとしてまとめる.

Stanford Basic アノテーションでは, 並列句構造はすべて右側の子供への依存構造として表されるため, ReduceLeft では大げさな処理は必要がなく, 簡単な処理となる.

なお, 6 行目で用いられている関数 car は, 引数として与えられた文字列 (S 式を表す) の最初の非終端記号を返す関数である. 例えば, car(’(ML R L)’) = ’ML’ となる.

ReduceRight

引数として与えられた節点 v の右側の子供を, v に近いものから順に, 手続き Traverse を呼ぶ. 子供 c の leftS, rightS として得られた結果を, v の導出ステップとしてまとめる.

Eisner の規則だけを用いた導出木への変換の際には, 本来 ReduceLeft と ReduceRight は対称性のある単純な手続きになるが, ここでの ReduceRight には, 並列構造の変換処理が加わっているため, ReduceLeft と非対称の, 比較的複雑な条件分岐をともなった処理が行われる.

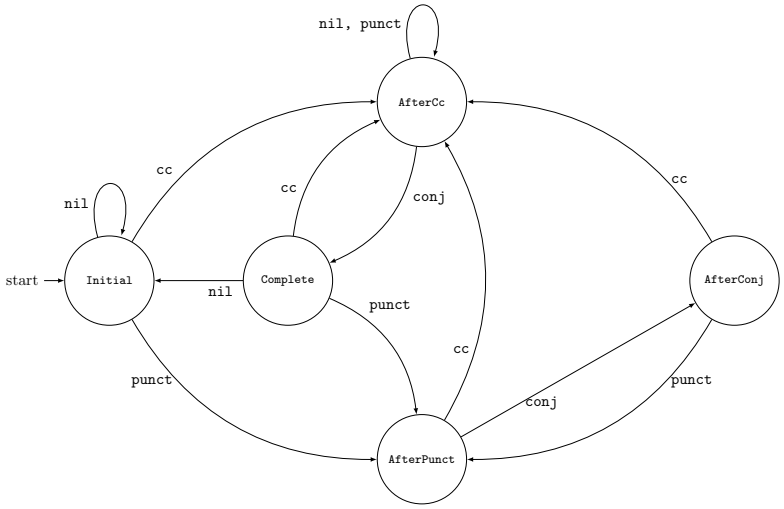


図 4.1: 状態遷移図

ここでは, 子供への辺の role の遷移が, 3 章の並列句構造処理用に導入した規則で扱えるものに合致してる, と仮定している. この遷移図を図 4.1 に示す.

手続き ReduceRight 左から右に子供をスキャンするが, その系列に応じた状態を state 変数に保持する.

擬似コード中, $AllowedRoles(state)$, $NextState(state, role[c])$ という関数が呼び出されているが (5 行目および 37 行目), これらはそれぞれ, 図 4.1 の状態遷移図の遷移が可能な状態集合 (AllowedRoles), 次状態 (NextState) を返す. 21, 28, 32 行目の関数 *car* については, 前節の ReduceLeft の解説で上述した通りである.

Algorithm 1: Global varibales

input : Word indice $V = \{1, \ldots, n\}$
input : Vocabulary W
input : Dependency types L
input : Word form function $form : V \mapsto W$
input : Head function $head : V \mapsto V \cup \{0\}$
input : Next sibling function $NextSibling : V \mapsto V$
input : Dependency type function $type : V \mapsto L$

Algorithm 2: Function *Convert* translates the dependency tree of a sentence to its corresponding derivation tree

```
1 function Convert()
      output: S-expression string representing the derivation tree
2     role ← MarkRoles()
3     foreach v ∈ V do
          /* Initialize the S-expressions representing the left and right descendants of each
             node                                                                          */
4         leftS[v] ← 'L' + str(v) + '/' + form(v)
5         rightS[v] ← 'R' + str(v) + '/' + form(v)
6     rightS[0] ← 'R0/<ROOT>'                    /* Root node (0) has rightS only */
7     Traverse(0, role, leftS, rightS)
8     return rightS[0]
```

Algorithm 3: Procedure *Traverse* walks over nodes in a dependency tree, translating dependency arcs into derivation steps along the way

```
1 procedure Traverse(v, role, var leftS, var rightS)
      input : Node v ∈ V
      input : Array of coordination roles for nodes
              role : V ↦ {cc, conj, punct, nil}
      input : Array of S-expression strings representing the derivation of the left
              descendants of a node leftS : V ↦ String
      input : Array of S-expression strings representing the derivation of the right
              descentants of a node rightS : V ∪ {0} ↦ String ∪ {nil}
2     ReduceLeft(v, role, leftS, rightS)
3     ReduceRight(v, role, leftS, rightS)
```

Algorithm 4: Procedure *ReduceLeft* traverses the left children of a node, nearest first

```
1  procedure ReduceLeft(v, role, var leftS, var rightS)
       input  : Node v ∈ V
       input  : Coordination role function role : V ↦ {cc, conj, punct, nil}
       input  : Array of S-expression strings representing left descendants of each
                node leftS : V ↦ String
       input  : Array of S-expression strings representing right descentants of each
                node rightS : V ∪ {0} ↦ String ∪ {nil}
2      Let C ← {c ∈ V | head[c] = v and c < v}
3      foreach c ∈ C, in descending order of c do   /* walk over left children of v, nearest
       first */
4          Traverse(c, role, leftS, rightS)
5          if rightS[c] = nil then                  /* if c is a coordination head */
6              if car(leftS[c]) ≠ 'CL' or car(leftS[v]) ≠ 'L' then
7                  throw UnexpectedNontermException
8              leftS[v] ← '(L ' + leftS[c] + ' ' + leftS[v] + ')'
9          else
10             leftS[v] ←
               '(L ' + leftS[c] + ' ' + '(ML ' + rightS[c] + ' ' + leftS[v] + '))'
```

Algorithm 5: Procedure *MakeConj*, a subroutine for building an S-expression for a conjunct

```
1 procedure MakeConj(v, nonterm, var leftS, var rightS)
      input : Node v ∈ V
      input : Nonterminal symbol (a string) nonterm
      input : Array of S-expression strings representing left descendants of each
              node leftS : V ↦ String
      input : Array of S-expression strings representing right descentants of each
              node rightS : V ∪ {0} ↦ String ∪ {nil}
2     if rightS[v] = nil then                          /* if v is a coordination head */
3         leftS[v] ← '(' + nonterm + ' ' + leftS[v] + ')'
4     else
5         leftS[v] ← '(' + nonterm + ' ' + leftS[v] + ' ' + rightS[v] + ')'
6         rightS[v] ← nil
```

4.2 既存の依存構造アノテーションからの正解構文木の作成

提案した規則を用いて実際に解析を行うためには，既存の依存構造アノテーションから正解構文木を作成できなければならない．

本節では実際に Penn Treebank を Stanford Basic アノテーションに変換したデータに対して 4.1 節で示したアルゴリズムを適用し，提案規則を用いて Stanford Basic アノテーションのスキームで解析できるものを確認した．

その結果，一部変換できない事例があったため，その傾向について述べる．

4.2.1 変換できなかった事例

Penn Treebank の Devset で変換できなかった事例で特殊な依存構造のものを，それぞれ図 4.2，図 4.3，図 4.4，図 4.5，図 4.6 に示す．図 4.2 は or の特殊な用法に関する事例，図 4.3 は CC の依存関係の方向が左向きになる例，図 4.4 は CC に対応する CONJ が存在しない例，図 4.5 と図 4.6 は CC の後に CONJ が 2 回出現する例である．

Penn Treebank の Devset で変換できなかった事例はこれらで全てである．Devset における全 1700 文中，並列構造が含まれるのは 725 文で，その中の 14 文が変換

Algorithm 6: Procedure *ReduceRight* visits the right children of a node, doing the necessary translation along the way. The results are stored in two arrays of S-expressions, *leftS* and *rightS*. This procdeure is not symmetric to *ReduceLeft* because of coordination processing it performs.

```
1  procedure ReduceRight(v, role, var leftS, var rightS)
       input  : Node v ∈ V
       input  : Coordination role function role : V ↦ {cc, conj, punct, nil}
       input  : Array of S-expression strings representing left descendants of each node leftS : V ↦ String
       input  : Array of S-expression strings representing right descentants of each node rightS : V ∪ {0} ↦ String ∪ {nil}
2      state ← Initial
3      C ← {c ∈ V | head[c] = v and c > v}                          /* C: children on the right of v */
4      foreach c ∈ C, in ascending order of c do                   /* walk over right children of v, nearest first */
5          if role[c] ∉ AllowedRoles(state) then throw UnexpectedRoleException
6          switch role[c] do
7              case punct
8                  Traverse(c, role, leftS, rightS)
9                  if state ≠ AfterConj then MakeConj(v, 'MC', leftS, rightS)
10                 leftS[v] ← '(IC ' + leftS[v] + ' (PM ' + leftS[c] + ' ' + rightS[c] + '))'
11             case cc
12                 Traverse(c, role, leftS, rightS)
13                 if state ≠ AfterConj then MakeConj(c, 'MC', leftS, rightS)
14                 leftS[v] ← '(NL ' + leftS[v] + ' (CM ' + leftS[c] + ' ' + rightS[c] + ')'
15             case conj
16                 Traverse(c, role, leftS, rightS)
17                 if state = AfterCc then
18                     MakeConj(c, 'LC', leftS, rightS)
19                     leftS[v] ← '(CL ' + leftS[v] + ' ' + leftS[c] + ')'
20                 else
21                     if rightS[v] ≠ nil or car(leftS[v]) ∉ {'CL', 'IC'} then
22                         throw UnexpectedNontermException
23                     MakeConj(c, 'BC', leftS, rightS)
24                     leftS[v] ← '(MC ' + leftS[v] + ' ' + leftS[c] + ')'
25             case nil                                            /* everything else */
26                 Traverse(c, role, leftS, rightS)
27                 if rightS[c] = nil then
28                     if car(rightS[v]) ≠ 'R' or car(rightS[v]) ≠ 'IC' then
29                         throw UnexpectedNontermException
30                     leftS[v] ← '(R ' + rightS[v] + ' ' + leftS[c] + ')'
31                 else if rightS[v] = nil then                    /* v is a coordination head but c is not */
32                     if car(leftS[v]) ≠ 'CL' or car(leftS[c]) ≠ 'L' then
33                         throw UnexpectedNontermException
34                     leftS[v] ← '(CL (RI ' + leftS[v] + ' ' + leftS[c] + ') ' + rightS[c] + ')'
35                 else                                            /* normal dependency arc */
36                     leftS[v] ← '(R (MR ' + rightS[v] + ' ' + leftS[c] + ' ' + rightS[v] + '))'
37         state = NextState(state, role[c])
```

Algorithm 7: Function *MarkRoles* to mark coordination roles to dependency arcs

```
1  function MarkRoles()
       output: Coordination role function role : V ↦ {cc, conj, punct, nil}
2      for v ∈ V do role[v] = nil
3      for v ∈ V do
4          C ← {c ∈ V | head[c] = v and c > v}                    /* right children of v */
5          r ← RightmostConj(C)   /* rightmost right child of v with type conj, or −∞ if
               none */
6          ℓ ← LeftmostConjCc(C)   /* leftmost right child of v with type conj, or +∞ if
               none */
7          foreach c ∈ C do
8              switch type[c] do
9                  case conj
10                     role[c] ← conj
11                 case cc
12                     if c < r then
13                         role[c] ← cc
14                     else
15                         throw IsolatedCcException
16                 case punct
17                     if form(c) ∈ {',', ';'} then
18                         if ℓ < c < r or type[NextSibling(c)] ∈ {cc, conj} then
19                             role[c] ← punct
20     return role
```

Algorithm 8: Function *RightmostConj* finds the node of type conj, having the largest index in a given set

```
1 function RightmostConj(C)
      input : Set of nodes C ⊂ V
2     if C contains a node c such that type[c] = conj then
3         return one such c ∈ C with the largest index
4     else
5         return −∞
```

Algorithm 9: Function *LeftmostConj* finds the node of type conj or cc, havingthe smallest index

```
1 function LeftmostConjCc(C)
      input : Set of nodes C ⊂ V
2     if C contains a node c such that type[c] ∈ {conj, cc} then
3         return one such c ∈ C with the smallest index
4     else
5         return +∞
```

cc: **coordination**
A coordination is the relation between an element of a conjunct and the coordinating conjunction word of the conjunct. (Note: different dependency grammars have different treatments of coordination. We take one conjunct of a conjunction (normally the first) as the head of the conjunction.) A conjunction may also appear at the beginning of a sentence. This is also called a cc, and dependent on the root predicate of the sentence.

conj: **conjunct**
A conjunct is the relation between two elements connected by a coordinating conjunction, such as “and”, “or”, etc. We treat conjunctions asymmetrically: The head of the relation is the first conjunct and other conjunctions depend on it via the conj relation.

表 4.2: Stanford typed dependencies manual （抜粋）

できなかったことになるが，それを除く文は全てこの手法で変換できることがわかった．

Stanford typed dependencies manual [De Marneffe and Manning, 2008] では，CC と CONJ について 4 ページに表 4.2 のように記されている．CC は conjunct の中の要素とその conjunct の中の CC との関係ラベルであり，conjunct は最初の conjunct が Head となり，他の conjunct はその conjunct の右側の CONJ となることがわかる．変換できなかった文に関して，図 4.3 のように CC の依存関係の方向が左向きになる例は，Stanford Basic アノテーションでは文頭が CC の場合を除いて考えにくく，Stanford 変換器のバグである可能性が考えられる．

4.2.2 変換された構文木に出現した規則

Penn Treebank に対して，4.1 節で示したアルゴリズムを適用した際に，作られた構文木に現れた規則を表 4.3 に示す．この規則は 1st order の Eisner 規則に表 3.1 の規則を加えた形になっている．このことから，このアルゴリズムによって 3 章で提案した規則を用いた構文木が作られることがわかる．

L	→	L ML
R	→	MR R
ML	→	R L
MR	→	R L
PUNCTMARK	→	L R
CCMARK	→	L R
BEFORECCCONJ	→	L R
BEFORECCCONJ	→	COORDCOMPLETE
LASTCONJ	→	L R
LASTCONJ	→	COORDCOMPLETE
MIDDLECONJ	→	L R
MIDDLECONJ	→	INCOMPLETECONJ
MIDDLECONJ	→	COORDCOMPLETE
MIDDLECONJ	→	INCOMPLETECONJ BEFORECCCONJ
INCOMPLETECONJ	→	MIDDLECONJ PUNCTMARK
NEXTLAST	→	MIDDLECONJ CCMARK
COORDCOMPLETE	→	NEXTLAST LASTCONJ
COORDCOMPLETE	→	RIGHTINCOMPLETE R
RIGHTINCOMPLETE	→	COORDCOMPLETE L
L	→	COORDCOMPLETE L
R	→	R COORDCOMPLETE

表 4.3: 変換アルゴリズムを適用した際に出現した規則

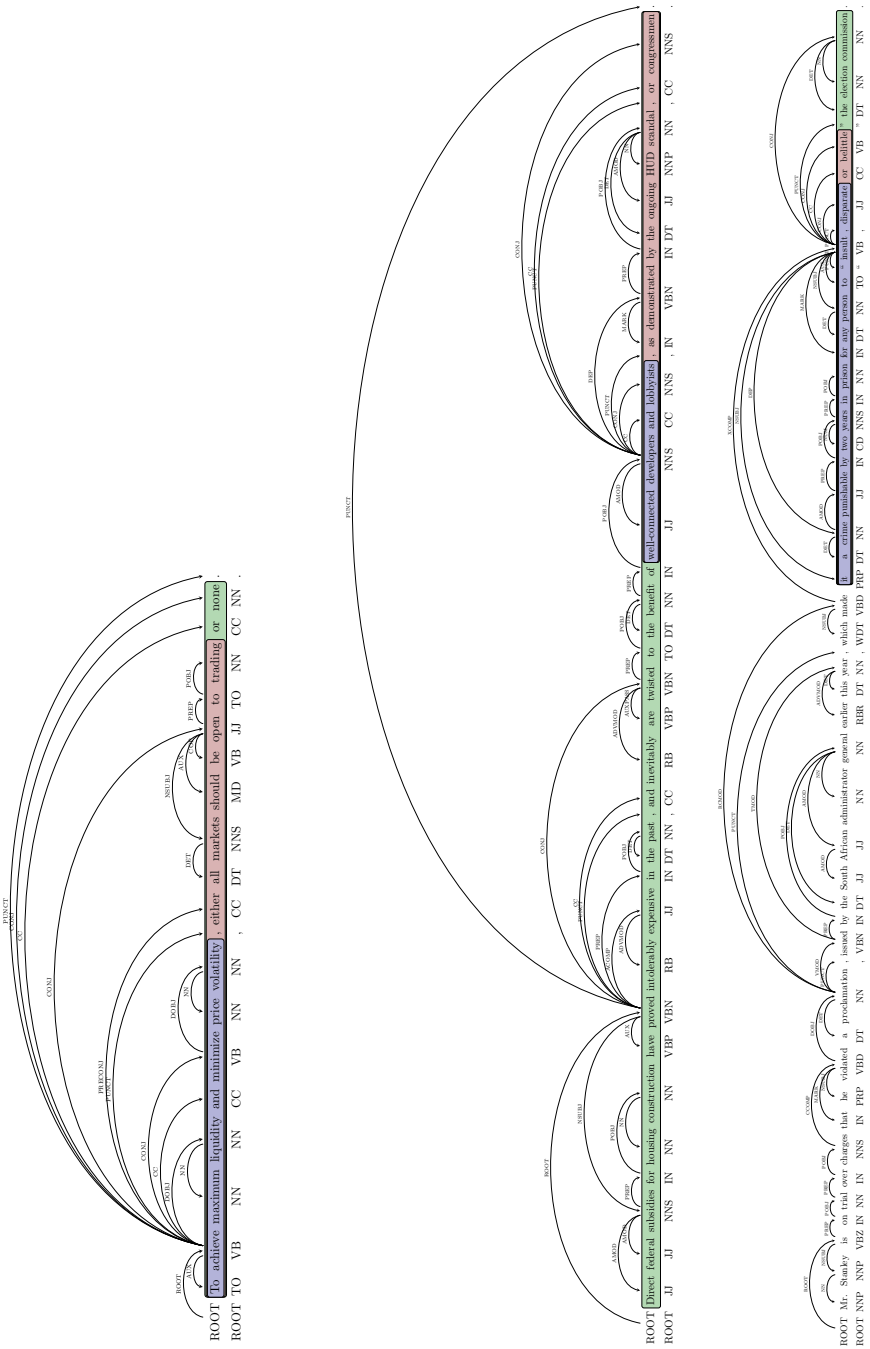


図 4.2: or の特殊な用法に関する事例

図 4.3: CC の依存関係の方向が左向きになる例

図 4.4: CC に対応する CONJ が存在しない例

図 4.5: CC の後に CONJ が 2 回出現する例 1

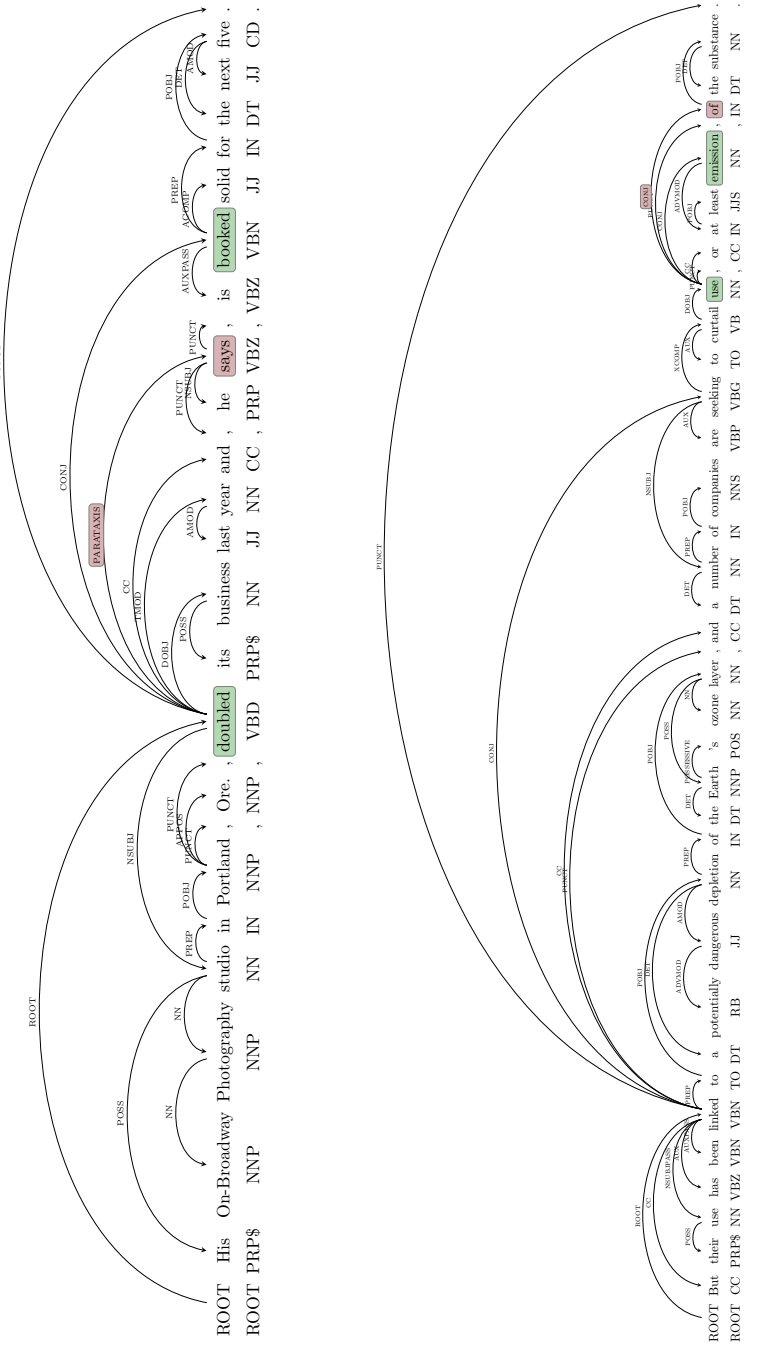


図 4.6: CC の後に CONJ が 2 回出現する例 2

第5章　結論

5.1　本研究の成果

本研究の成果は以下の 3 点である.

1. 依存構造解析の過程で並列構造を捉えるための規則の提案
2. 既存の依存構造アノテーションからの機械的な変換手法の提案
3. 並列構造のパターンの分類

本研究では，並列構造解析を構文解析器，特に依存構造解析器において，並列構造を他の依存構造の解析と同時に処理する手法の第一歩として，並列構造を捉えるために必要な推論規則を提案した.

並列構造解析のためには，並列構造をなす各並列項目の範囲を依存構造解析の各段階で捉える必要があり，本来の Eisner アルゴリズムでは並列項目の範囲が解析の各段階で分断されているために捉えられなかった.

Eisner アルゴリズムの規則に，本提案規則を組み合わせることにより，並列項目の候補となる範囲において並列項目候補の範囲の分断を防ぎ，各並列項目候補同士の比較が可能になる.

しかし実際に依存構造解析器を作成するためには，依存構造が人手でアノテーションされたコーパスから正解構文木を作成し，その構文木を組み立てるために役立つ手がかりを機械学習によって捉えなければならない.

そこで本研究では Stanford アノテーションから，提案規則を用いた構文木を自動的に構築するアルゴリズムも提案し，実際に Devset において構文木を構築できることを示した.

自動的に正解構文木を構築できるようにしたことによって，人手によるコストをかけずに，既存のアノテーションによるコーパスから，提案手法を用い，並列構造解析を含んだ依存構造解析器の機械学習による訓練が可能になる.

5.2 展望

本研究では並列構造解析のための規則を提案し，実際に構文木が構築できることを示した．今後の課題として，本手法を用いた並列構造解析を含む構文解析器を作成し，並列構造解析を含まない解析器に比べて本手法が有効であることを確認する予定である．

参考文献

[1] Kazuo Hara, Masashi Shimbo, Hideharu Okuma, and Yuji Matsumoto. Coordinate structure analysis with global structural constraints and alignment-based local features. In *Proceedings of the 47th Annual Meeting of the ACL and the 4th IJCNLP of the AFNLP*, pages 967–975, 2009.

[2] Atsushi Hanamoto, Takuya Matsuzaki, and Jun'ichi Tsujii. Coordination structure analysis using dual decomposition. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 430–438. Association for Computational Linguistics, 2012.

[3] Truc-Vien T Nguyen, Alessandro Moschitti, and Giuseppe Riccardi. Convolution kernels on constituent, dependency and sequential structures for relation extraction. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3-Volume 3*, pages 1378–1387. Association for Computational Linguistics, 2009.

[4] Jason Katz-Brown, Slav Petrov, Ryan McDonald, Franz Och, David Talbot, Hiroshi Ichikawa, Masakazu Seno, and Hideto Kazawa. Training a parser for machine translation reordering. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 183–192. Association for Computational Linguistics, 2011.

[5] Jun Xie, Haitao Mi, and Qun Liu. A novel dependency-to-string model for statistical machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 216–226. Association for Computational Linguistics, 2011.

[6] Kazuo Hara and Masashi Shimbo. A discriminative learning model for coordinate conjunctions. In *EMNLP-CoNLL*, pages 610–619, 2007.

[7] Jason M Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, pages 340–345. Association for Computational Linguistics, 1996.

[8] Dan Klein and Christopher D Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in neural information processing systems*, pages 3–10, 2002.

[9] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[10] Mark Johnson. Transforming projective bilexical dependency grammars into efficiently-parsable cfgs with unfold-fold. In *Proceedings of the 45th annual meeting of the Association of Computational Linguistics*, volume 45, pages 168–175, 2007.

[11] Daniel H Younger. Recognition and parsing of context-free languages in time n^3. *Information and control*, 10(2):189–208, 1967.

[12] Richard Johansson and Pierre Nugues. Extended constituent-to-dependency conversion for English. In *Proceedings of NODALIDA 2007*, pages 105–112, Tartu, Estonia, May 25-26 2007.

[13] Philip Resnik. Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language. *Journal of Artificial Intelligence Research*, 11:95–130, 1999.

[14] Preslav Nakov and Marti Hearst. Using the web as an implicit training set: application to structural ambiguity resolution. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 835–842. Association for Computational Linguistics, 2005.

[15] Francis Chantree, Adam Kilgarriff, Anne De Roeck, and Alistair Willis. Disambiguating coordinations using word distribution information. *Proceedings of RANLP2005*, 2005.

[16] Rajeev Agarwal and Lois Boggess. A simple but useful approach to conjunct identification. In *Proceedings of the 30th annual meeting on Association for Com-*

putational Linguistics, pages 15–21. Association for Computational Linguistics, 1992.

[17] Deirdre Hogan. Coordinate noun phrase disambiguation in a generative parsing model. In *Proceedings of the 45th annual meeting of the Association of Computational Linguistics*, volume 45, page 680, 2007.

[18] Akitoshi Okumura and Kazunori Muraki. Symmetric pattern matching analysis for english coordinate structures. In *Proceedings of the fourth conference on Applied natural language processing*, pages 41–46. Association for Computational Linguistics, 1994.

[19] Ekaterina Buyko, Katrin Tomanek, and Udo Hahn. Resolution of coordination ellipses in biological named entities using conditional random fields. In *PACLING 2007-Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, pages 163–171, 2007.

[20] Ekaterina Buyko and Udo Hahn. Are morpho-syntactic features more predictive for the resolution of noun phrase coordination ambiguity than lexico-semantic similarity scores? In *Proceedings of the 22nd International Conference on Computational Linguistics-Volume 1*, pages 89–96. Association for Computational Linguistics, 2008.

[21] Daniel M Bikel. Design of a multi-lingual, parallel-processing statistical parsing engine. In *Proceedings of the second international conference on Human Language Technology Research*, pages 178–182. Morgan Kaufmann Publishers Inc., 2002.

[22] Sadao Kurohashi and Makoto Nagao. A syntactic analysis method of long japanese sentences based on the detection of conjunctive structures. *Computational Linguistics*, 20(4):507–534, 1994.

[23] 黒橋禎夫. 日本語構文解析システム knp version 2.0 b6 使用説明書. **京都大学大学院 情報学研究科**, 6, 1998.

[24] 黒橋禎夫. 結構やるな, knp. **情報処理**, 41(11):1215–1220, 2000.

[25] Dan Gusfield. Algorithms on strings, trees, and sequences. *ACM SIGACT News*, 28(4):41–60, 1997.

[26] Carl Pollard. *Head-driven phrase structure grammar*. University of Chicago Press, 1994.

[27] Yusuke Miyao and Jun'ichi Tsujii. Deep linguistic analysis for the accurate identification of predicate-argument relations. In *Proceedings of the 20th international conference on Computational Linguistics*, page 1392. Association for Computational Linguistics, 2004.

[28] Sadao Kurohashi and Makoto Nagao. Dynamic programming method for analyzing conjunctive structures in japanese. In *Proceedings of the 14th conference on Computational linguistics-Volume 1*, pages 170–176. Association for Computational Linguistics, 1992.

[29] Marie-Catherine De Marneffe and Christopher D Manning. Stanford typed dependencies manual. *URL http://nlp. stanford. edu/software/dependencies manual. pdf*, 2008.