Parsing Dependencies

Abstract

Motivated by the increase in accuracy of statistical dependency parsers, we consider the problem of decoding phrase-structure parses directly from predicted dependency trees. Unlike past rule-based approaches, we treat this as a structured prediction problem and train a constrained context-free parser for this task. Since our parser is constrained by the dependency structure it is both asymptotically and empirically faster than standard lexicalized parsers. However, despite its simplicity, it still yields high-accuracy phrase-structure parses on experiments in both English and Chinese.

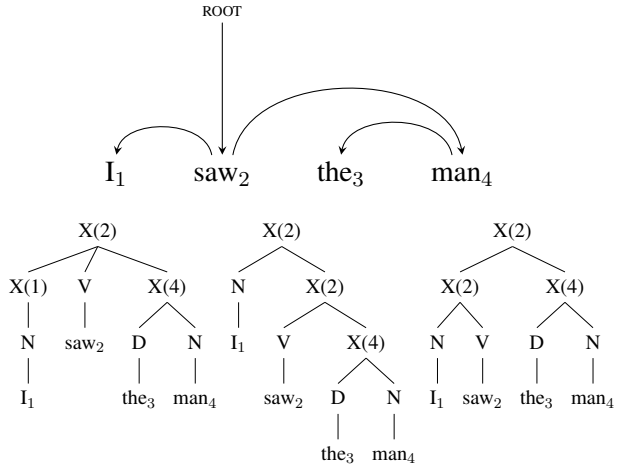


Figure 1: While a phrase-structure parse determines a unique dependency parse, the inverse problem is non-deterministic. The figure, adapted from (Collins et al., 1999), shows several X-bar trees that all produce the same dependency structure. The parentheses $X(h)$ indicate the head h of each internal vertex.

1 Introduction

There are two dominant grammatical frameworks used for statistical syntactic parsing: phrase-structure and dependency parsing (). The two formalisms offer a trade-off. Phrase-structure parsing is very accurate and provides a full context-free grammatical representation; while dependency parsing is much faster, both asymptotically and empirically, while still predicting much of the important structural relationships in a sentence.

Recent advances in dependency parsing have lead to models that perform nearly as well as phrase-structure parsers in terms of dependency accuracy (), and (**?**) Table 1 shows that for Collins head rules (), the two state-of-the-art dependency parsers score only % below the best phrase-structure parsers.

-

However, dependencies alone provide less value than a full phrase-structure parse, and many applications still rely on having the phrase-structure (). But while the transformation from phrase-structure to dependencies is deterministic, the inverse is not, as illustrated in Figure 1.

We explore the problem of recovering the phrase-structure from the dependency representation. This recovery is challenging for several reasons: (a) there are an exponential number of phrase-structure trees for any dependency parse, (b) in addition to the tree structure, it is necessary to recover the non-terminal symbols of the tree, and (c) the parser must be robust to errors in the downstream predicted dependency tree.

We treat this task as a structured prediction prob-

lem, and train a complete phrase-structure parser to predict the syntactic tree for a given sentence. However, we limit the search space of the parser to the skeleton of a given dependency tree. Experiments show that

- The constrained parser is asymptotically faster than standard phrase-structure parser for lexicalized context-free grammar. A standard algorithm for this problem is $O(n^5|\mathcal{R}|)$, but with constrained dependency structure the worst case $isO(n^2|\mathcal{R}|)$.
- In practice using simple pruning that parser is linear time in the length of the sentence and as efficient as the fastest high-accuracy dependency parsers. %
- Despite being constrained to hard downstream dependency decisions, the parser is comparably accurate to non-reranked phrase-structure parsers. %

The problem of converting dependency to phrase-structured trees has been studied previously from a treebanking perspective. Xia and Palmer (2001) and Xia et al. (2009) develop a rule-based system for the converting human-annotated dependency parses. In this work we learn the conversion from data, and consider the case of automatically predicted parses.

This task is similar to phrase-structure parsers that utilize dependency parsing techniques. Carreras et al. (2008) build a high-accuracy parser that uses a dependency parsing model both for pruning and within a richer lexicalized parser. Similarly Rush et al. (2010) use dual decomposition to combine a dependency parser with a simple phrase-structure model. However, we take this approach a step further by fixing the dependency structure entirely.

Finally there have also been several papers that use ideas from dependency parsing to simplify and speed up phrase-structure prediction. Zhu et al. (2013) build a high-accuracy phrase-structure parser using a transition-based system. Hall et al. (2014) use a stripped down parser based on a simple X-bar grammar and a small set of lexicalized features.

2 Background

We begin by developing notation for a lexicalized phrase-structure formalism and for dependency parsing. The notation aims to highlight the similarity between the two formalisms.

Model	sec 22 UAS	oracle score	speed
Berkeley	$[O(n^5)]$	100%	speed
stanford TurboParse	$[O()]$	oracle	

Table 1: Collins Head Rules scores on Dev.

2.1 Lexicalized CFG Parsing

A lexicalized context-free grammar (LCFG) is an extended context-free grammar where each vertex in a parse has a unique lexical head word. Define an binarized LCFG as a 4-tuple $(\mathcal{N}, \mathcal{R}, \mathcal{T}, r)$ where:

- $\mathcal{N}$; a set of nonterminal symbols, e.g. `NP`, `VP`.
- $\mathcal{T}$; a set of terminal symbols, often consisting of the words in the language.
- $\mathcal{R}$; a set of lexicalized rule productions either of the form $A \rightarrow \beta_1^* \ \beta_2$ or $A \rightarrow \beta_1 \ \beta_2^*$ consisting of a parent nonterminal $A \in \mathcal{N}$, a sequence of children $\beta_i \in \mathcal{N}$ for $i \in \{1 \dots 2\}$, and a distinguished head child annotated with $*$. The head child comes from the head rules associated with the grammar.
- r; a distinguished root symbol $r \in \mathcal{N}$.

Given an input sentence $x_1, \dots, x_n$ of terminal symbols from $\mathcal{T}$, define $\mathcal{Y}(x)$ as the set of valid lexicalized parses for the sentence. This set consists of all binary ordered trees with fringe $x_1, \dots, x_n$, internal nodes labeled from $\mathcal{N}$, all tree productions $A \rightarrow \beta$ consisting of members of $\mathcal{R}$, and root label r.

For an LCFG parse $y \in \mathcal{Y}(x)$, we further associate a triple $v = (\langle i, j \rangle, h, A)$ with each vertex in the tree, where

- $\langle i, j \rangle$; the *span* of the vertex, i.e. the contiguous sequence $\{x_i, \dots, x_j\}$ of the sentence covered by the vertex.
- $h(v) \in \{1, \dots n\}$; index indicating that x_h is the *head* of the vertex, defined recursively by the following rules:

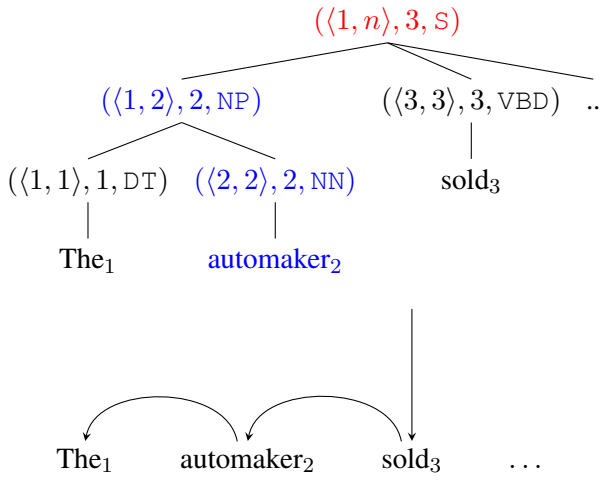


Figure 2: Figure illustrating an LCFG parse. The parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue vertices represent the 3-vertex spine v_1, v_2, v_3 of the word $\texttt{automaker}_2$. The root vertex is v_4, which implies that $\texttt{automaker}_2$ modifies $\texttt{sold}_3$ in the induced dependency graph.

1. If the vertex is leaf x_i, then $h = i$.
2. Otherwise, h matches the head child where $A \rightarrow \beta_1^* \ \beta_2$ or $A \rightarrow \beta_1 \ \beta_2^*$ is the rule production at this vertex.

- $A \in \mathcal{T} \cup \mathcal{N}$; the terminal or nonterminal symbol of the vertex.

Note that each word x_i but one has an ancestor vertex v where $h(v) \neq i$. Define the *spine* of word x_i to be the longest chain connected vertices $v_1, \ldots, v_p$ where $h(v_j) = i$ for $j \in \{1, \ldots, p\}$. Also if it exists, let vertex v_{p+1} be the parent of vertex v_p, where $h(v_{p+1}) \neq i$. The full notation is illustrated in Figure 2.

2.2 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of grammatical structure.

Given an input sentence $x_1 \ldots x_n$, define a dependency parse d as a sequence $d_1 \ldots d_n$ where for all i, $d_i \in \{0, \ldots, n\}$. These dependency relations can be seen as arcs (d_i, i) in a directed graph over the word indices where 0 is a special pseudo-root vertex. A dependency parse is valid if the corresponding directed graph is a directed tree rooted at vertex 0. Figure 2 contains an example of a dependency tree.

For a valid dependency tree, define the *span* of any word x_m as the set of indices reachable from vertex m in the directed tree. A dependency parse is *projective* if the descendants of every word in the tree form a contiguous span of the original sentence (). We use the notation $m_{\Leftarrow}$ and $m_{\Rightarrow}$ to represent the left- and right-boundaries of this span.

Any lexicalized context-free parse can be converted to a unique projective dependency tree. For an input symbol x_m with spine $v_1, \ldots, v_p$,

1. If v_p is the root of the tree, then $d_m = 0$.
2. Otherwise let v_{p+1} be the parent vertex of v_p and $d_m = h(v_{p+1})$. The span $\langle i, j \rangle$ of v_p in the lexicalized parse is equivalent to $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ in the induced dependency parse.

The conversion produces a directed tree rooted by preserving the tree structure of the original LCFG parse.

However the reverse conversion is not unique, and in fact, it can be shown that in the worst-case there are an exponential number of possible phrase structure trees that match a given dependency parse. We give the proof in Appendix A.

3 Parsing Dependencies

Our main problem of interest is to predict the best phrase-structure tree for a given dependency parse. We frame this problem as a combinatorial decoding problem. First, we describe the standard lexicalized CKY algorithm and then give a constrained version of this algorithm.

3.1 Constrained Parsing Algorithm

For a given lexicalized context-free grammar, define the set of valid parses for a sentence as $\mathcal{Y}(x)$. The parsing problem is to find the highest-scoring parse in this set, i.e.

$$\hat{y} \leftarrow \arg\max_{y \in \mathcal{Y}(x)} s(y; x)$$

where s is a scoring function.

If the scoring function factors over rule productions, then the highest-scoring parse can be found

using the lexicalized CKY algorithm. This algorithm is defined as a collection of inductive rules shown in Figure 3.1.

For example consider the inductive rule

$$\frac{(\langle i, k\rangle, m, \beta_1) \quad (\langle k+1, j\rangle, h, \beta_2)}{(\langle i, j\rangle, h, A)}$$

for all rules $A \rightarrow \beta_1^* \ \beta_2 \in \mathcal{R}$ and spans $i \leq k < j$. This rule indicates that grammatical rule $A \rightarrow \beta_1^* \ \beta_2$ was applied at a vertex covering $\langle i, j\rangle$ to produce two vertices covering $\langle i, k\rangle$ and $\langle k+1, j\rangle$.

The best parse can found by bottom-up dynamic programming over this set. The running time of this algorithm is linear in the number of rules. The standard algorithm requires $O(n^5|\mathcal{R}|)$ time, which is intractable to run without heavy pruning.

However, in this work, we are interested in a constrained variant of this problem. We assume that we additionally have access to a projective dependency parse for the sentence, $d_1, \ldots, d_n$. Define the set $\mathcal{Y}(x, d)$ as all valid LCFG parses that match this dependency parse. For all inductive rules with head h and modifier m, there must be a dependency (h, m) in d. Our aim is to find

$$\arg\max_{y \in \mathcal{Y}(x,d)} s(y; x, d)$$

This new problem has a nice property. For any word x_m with spine $v_1 \ldots v_p$ the LCFG span $\langle i, j\rangle$ of v_p is equal to the dependency span $\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle$ of x_m. Furthermore these spans can be easily computed directly from given d.

This property greatly limits the search space of the parsing problem. Instead of searching over all possible spans $\langle i, j\rangle$ of each modifier, we can precompute $\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle$. Figure 3.1 shows the new set inductive rules.

While these rules are very similar to the original, the quantifier are much more constrained. Given that there are n dependency links (h, m) and n indices, the new algorithm has $O(n^2|\mathcal{R}|)$ running time.

3.2 Extension: Labels

Finish this section

Standard dependency parsers also predict labels from a set $\mathcal{L}$ on each dependency link. In a labeled dependency parser a would be of the form (i, j, l).

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \ldots n\}, A \in \mathcal{N}$$

Rules:

For $i \leq h \leq k < m \leq j$, and rule $A \rightarrow \beta_1^* \ \beta_2$,

$$\frac{(\langle i, k\rangle, h, \beta_1) \quad (\langle k+1, j\rangle, m, \beta_2)}{(\langle i, j\rangle, h, A)}$$

For $i \leq m \leq k < h \leq j$, rule $A \rightarrow \beta_1 \ \beta_2^*$,

$$\frac{(\langle i, k\rangle, m, \beta_1) \quad (\langle k+1, j\rangle, h, \beta_2)}{(\langle i, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m$$

Figure 3: Standard CKY algorithm for LCFG parsing stated as inductive rules. Starting from the *premise*, any valid application of *rules* that leads to a *goal* is a valid parse. Finding the optimal parse with dynamic programming is linear in the number of rules. For this algorithm there are $O(n^5|\mathcal{R}|)$ rules where n is the length of the sentence.

Model	% items pruned	oracle score	speed
LCFG	$[O(n^5)]$	100%	speed
Dep Parse Model	$[O()]$	oracle	
Pruned Model	$O(n)$	pruned oracle	

Table 2: Pruning

This label information can be used to encode further information about the parse structure. For instance if we use the label set $\mathcal{L} = \mathcal{N} \times \mathcal{N} \times \mathcal{N}$, encoding the binary rule decisions $A \rightarrow \beta_1 \ \beta_2$.

3.3 Extension: Pruning

Pruning discussion

3.4 Binarization

While the algorithm itself is not dependent on the LCFG binarization method used, the choice of binarization effects the run-time of the algorithm, through $\mathcal{R}$, as well as the modeling accuracy, through the factored scoring function s.

Since the parser traces a fixed dependency struc-

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:
For all $i < m, h = d_m$ and rule $A \rightarrow \beta_1^* \ \beta_2$,

$$\frac{(\langle i, m_{\Leftarrow} - 1\rangle, h, \beta_1) \quad (\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_2)}{(\langle i, m_{\Rightarrow}\rangle, h, A)}$$

For all $m < j, h = d_m$ and rule $A \rightarrow \beta_1 \ \beta_2^*$,

$$\frac{(\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_1) \quad (\langle m_{\Rightarrow} + 1, j\rangle, h, \beta_2)}{(\langle m_{\Leftarrow}, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m \text{ s.t. } d_m = 0$$

Figure 4: The constrained CKY parsing algorithm for $\mathcal{Y}(x, d)$. The algorithm is nearly identical to Figure 3.1 except that many of the free indices are now fixed to the dependency parse. Finding the optimal parse is now $O(n^2|\mathcal{R}|)$
.

ture we select a binarization based around the head structure.

For simplicity, we consider binarizing rule $\langle A \rightarrow \beta_1 \dots \beta_m, k\rangle$ with $m > 2$. Relative to the head β_k the rule has left-side $\beta_1 \dots \beta_{k-1}$ and right-side $\beta_{k+1} \dots \beta_m$.

We replace this rule with binary rules that consume each side independently as a first-order Markov chain (horizontal Markovization). The main transformation is to introduce rules

- $A^{\Rightarrow}_{\beta_i} \rightarrow A^{\Rightarrow}_{\beta_{i-1}}{}^* \ \beta_i$ for $k > i > m$
- $A^{\Leftarrow}_{\beta_i} \rightarrow \beta_i \ A^{\Leftarrow}_{\beta_{i+1}}{}^*$ for $1 < i < k$

Additionally we introduce several additional rules to handle the boundary cases of starting a new rule, finishing the right side, and completing a rule. (These rules are slightly modified when $k = 1$ or $k = m$) .

- $A^{\Rightarrow}_{\beta_{k+1}} \rightarrow \beta_k^* \ \beta_{k+1}$
- $A^{\Rightarrow}_{\text{END}} \rightarrow A^{\Rightarrow}_{\beta_{m-1}}{}^* \ \beta_m$
- $A^{\Leftarrow}_{\beta_{k-1}} \rightarrow \beta_{k-1} \ A^{\Rightarrow}_{\text{END}}{}^*$
- $A \rightarrow \beta_1 \ A^{\Leftarrow *}_{\beta_2}$

Each rule contains at most 3 nonterminals so the size of the new binarized rule set is bounded by $O(\mathcal{N}^3)$.

4 Structured Prediction

To learn the scoring function for the transformation from dependency trees to phrase-structure trees, we use a standard structured prediction setup. We define the scoring function s as

$$s(y; x, d, \theta) = \theta^\top f(x, d, y)$$

where $\theta \in \mathbb{R}^D$ is a weight vector and $f(x, d, y)$ is a feature function that maps parse production (as in Figure **??**) to sparse feature vectors in $\{0, 1\}^D$. In this section we first discuss the features used and then training for the weight vector.

4.1 Features

We implemented a small set of standard dependency and phrase-structure features.

For the dependency style features, we replicated the basic arc-factored features used by McDonald (2006). These include combinations of:

For a part $\dfrac{(\langle i,k\rangle, m, \beta_1) \quad (\langle k+1,j\rangle, h, \beta_2)}{(\langle i,j\rangle, h, A)}$

Nonterm Features
(A, β_1)
(A, β_2)
$(A, \beta_1, \text{tag}(m))$
$(A, \beta_2, \text{tag}(h))$

Span Features
(rule, x_i)
(rule, x_j)
(rule, x_{i-1})
(rule, x_{j+1})
(rule, x_k)
(rule, x_{k+1})
$(\text{rule}, \text{bin}(j-i))$

Rule Features
(rule)
$(\text{rule}, x_h, \text{tag}(m))$
$(\text{rule}, \text{tag}(h), x_m)$
$(\text{rule}, \text{tag}(h), \text{tag}(m))$
(rule, x_h)
$(\text{rule}, \text{tag}(h))$
(rule, x_m)
$(\text{rule}, \text{tag}(m))$

Figure 5: The feature templates used in the function $f(x, d, y)$. The symbol rule is expanded into two conjunction $A \rightarrow B\ C$ and A. The function $\text{tag}(i)$ gives the part-of-speech tag of word x_i. The function $\text{bin}(i)$ bins a span length into 10 bins.

- nonterminal combinations
- rule and top nonterminal
- modifier word and part-of-speech
- head word word and part-of-speech

Additionally we included the span features described for the X-Bar style parser of Hall et al. (2014). These include conjunction of the rule with:

- first and last word of current span.
- preceding and following word of current span
- adjacent words at split of current span
- length of the span

The full feature set is shown in Figure 4.1. After training there are # non-zero features.

4.2 Training

We train the parameters θ using standard structured SVM training ().

We assume that we are given a set of gold-annotated parse examples: $(x^1, y^1), \ldots, (x^D, y^D)$. We also define $d^{(1)} \ldots d^{(D)}$ as the dependency structures induced from $y^1 \ldots y^D$. We select parameters to minimize the regularized empirical risk

$$\min_\theta \sum_{i=1}^{D} \max\{0, \ell(x^i, d^i, y^i, \theta)\} + \frac{\lambda}{2}||\theta||_1$$

where we define ℓ as

$$\ell(x, d, y, \theta) = s(y) + \max_{y' \in \mathcal{Y}(x,d)} \left(s(y') + \Delta(y, y')\right)$$

where Δ is a problem specific cost-function that we assume is linear in either arguments. In experiments, we use a hamming loss $\Delta(y, \bar{y}) = ||y - \bar{y}||$ where y is an indicator of rule productions.

The objective is optimized using Adagrad (). The gradient calculation requires computing a loss-augmented argmax for each training example which is done using the algorithm of Figure **??**.

5 Data and Setup

5.1 Data

We used wsj..

We used ctb 5-1..

At training time, we run 10-fold jack-knifing to produce dependency parses $\hat{d}$. We then run a single pass to calculate $\bar{y}$ for each training example.

5.2 Implementation

We built...

Parser is in C++, publicly available, 500 lines of code..

5.3 Binarization

Before describing our parsing algorithm we first describe a binarization approach to make efficient parsing possible and highlight the relationship between the LCFG and the dependency parse.

5.4 Extension: Pruning

We also experiment with a simple pruning dictionary pruning technique. For each context-free rule $A \rightarrow \beta_1\ \beta_2$ and POS tag a we remove all rules that were not seen with that tag as a head in training.

Parsing Results		
	wsj	
	speed	fscore
Petrov		
Carreras		
	ctb	

Table 3: This is the big monster result table that should tower above all comers.

Model	fscore	speed
TURBOPARSER		
MALTPARSER		
EASYFIRST		

Table 4: This

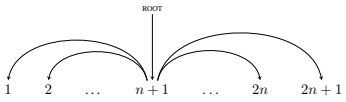
5.5

6 Results

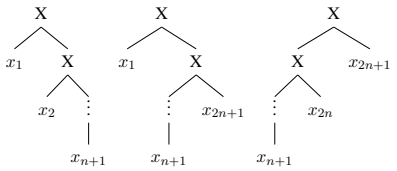
6.1

A Proof of PS Size

Consider the LCFG grammar with two rules $A = X \rightarrow X^* \; X$ and $B = X \rightarrow X \; X^*$ and a sentence $x_1, \ldots, x_{2n+1}$. Let the dependency parse be defined as $d_{n+1} = 0$ and $d_i = n + 1$ for all $i \neq n + 1$, i.e.



Since all rules have $h = x_n$ as head, a parse is a chain of $2n$ rules with each rule in $\{A, B\}$, e.g. the following are BB..., BA..., AA...



Since there must be equal As and Bs and all orders are possible, there are $\binom{2n}{n}$ valid parses and $|\mathcal{Y}(x, d)|$ is $O(2^n)$.

References

Xavier Carreras, Michael Collins, and Terry Koo. 2008. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 9–16. Association for Computational Linguistics.

Michael Collins, Lance Ramshaw, Jan Hajič, and Christoph Tillmann. 1999. A statistical parser for czech. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 505–512. Association for Computational Linguistics.

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *ACL*.

Ryan McDonald. 2006. *Discriminative learning and spanning tree algorithms for dependency parsing*. Ph.D. thesis, University of Pennsylvania.

Alexander M Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11. Association for Computational Linguistics.

Fei Xia and Martha Palmer. 2001. Converting dependency structures to phrase structures. In *Proceedings of the first international conference on Human language technology research*, pages 1–5. Association for Computational Linguistics.

Fei Xia, Owen Rambow, Rajesh Bhatt, Martha Palmer, and Dipti Misra Sharma. 2009. Towards a multi-representational treebank. In *The 7th International Workshop on Treebanks and Linguistic Theories. Groningen, Netherlands*.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443.