Transition-based dependency parsing as latent-variable constituent parsing

Mark-Jan Nederhof
School of Computer Science
University of St Andrews, UK

Abstract

We provide a theoretical argument that a common form of projective transition-based dependency parsing is less powerful than constituent parsing using latent variables. The argument is a proof that, under reasonable assumptions, a transition-based dependency parser can be converted to a latent-variable context-free grammar producing equivalent structures.

1 Introduction

Over the last decade, transition-based dependency parsers have received much attention, to a large extent due to Nivre (2003), Nivre and Scholz (2004), Nivre et al. (2004) and following publications. The theory represented in these publications seems to differ significantly from traditional automata theory, on which the theory of constituent parsing is based. Differences lie in notation, in terminology and in the overall conceptual framework.

An explanation for this cannot immediately be found by contrasting the foundations of dependency parsing and constituent parsing. Some of the earliest literature on dependency parsing (Hays, 1964; Gaifman, 1965) discusses the two kinds of parsing on an equal footing. Also more recent literature (Carroll and Charniak, 1992; Klein and Manning, 2004) discusses dependency parsing as closely related to constituent parsing. The concept of bilexical context-free grammars (Eisner and Satta, 1999) establishes further explicit connections between phrase-structure grammar and dependency grammar. See also Rambow (2010) for a discussion about the relation between constituent and dependency structures.

One advantage of dependency grammar is the ease with which the definition of parse structures can be generalized from the projective case to the non-projective case, but also this cannot explain the divergence from the theory of constituent parsing, as much the same style is used for describing projective dependency parsing and for non-projective dependency parsing; cf. Nivre (2009) for the latter. Furthermore, discontinuity has also been explored for constituent parsing (Kallmeyer and Maier, 2010; van Cranenburgh et al., 2011). Close links between discontinuous constituent parsing and non-projective dependency parsing follow from the work of, among others, Kallmeyer and Kuhlmann (2012) and Kuhlmann (2013).

Recent literature on dependency parsing has a strong emphasis on parsing speed. Often, parsing algorithms are close to linear-time, or close to quadratic-time in the worst case (Covington, 2001). However, there is also a considerable body of literature on speeding up constituent parsing (Lavie and Tomita, 1993; Goodman, 1997; Caraballo and Charniak, 1998). Deterministic parsing algorithms for constituent parsing were proposed by e.g. Wong and Wu (1999), Kalt (2004), Sagae and Lavie (2005) and Nederhof and McCaffery (2014), while the parser of Ratnaparkhi (1997) is close to linear time; for deterministic chunk parsing, see Tsuruoka and Tsujii (2005). Seneff (1989) suggests that deterministic constituent parsing was more or less the norm at the end of the 1980s. Conversely, transition-based dependency parsing has been generalized to non-deterministic parsing (Kuhlmann et al., 2011; Huang and Sagae, 2010).

An empirical connection between constituent parsing and dependency parsing has been established by several investigations of conversion between constituent structures and dependency structures. Transformation from constituent structures to dependency structures is addressed by Lin (1998), Collins (2003), Yamada and Matsumoto

(2003) and Hall and Novák (2005). Dependency parsers have been used to perform constituent parsing (Ma et al., 2010). Transformations from unlabeled dependency structures to constituent structures were discussed by Johnson (2007), and transformations from labeled dependency structures were discussed by Miyao et al. (2008). It has been observed that constituent parsers used to perform dependency parsing can be at least as accurate as dedicated dependency parsers, although they are generally slower (Cer et al., 2010; Candito et al., 2010).

The present article aims to elucidate part of the relation between the theory of transition-based dependency parsing and the theory of constituent parsing. We will focus on a particular form of constituent parsing that is based on latent-variable probabilistic context-free grammars, which currently offers state-of-the-art accuracy. One apparent complication is that there are competing ways of obtaining such grammars, roughly divided into forms of EM training (Matsuzaki et al., 2005; Petrov et al., 2006) or of spectral learning (Narayan and Cohen, 2015). We circumvent this complication by looking at the general class of *non-probabilistic* latent-variable context-free grammars, and show that these have sufficient formal power to subsume deterministic transition-based dependency parsing. The implication is that latent-variable *probabilistic* context-free grammars, obtained through EM training, spectral learning, or any other method still to be developed, have the potential to be at least as accurate as deterministic transition-based dependency parsing.

This paper intentionally limits the scope to projective parsing. The reason is that the literature on non-projectivity (discontinuity) has not yet converged, and new approaches are discovered with some regularity. This makes it hard to offer formal evidence that non-projective dependency parsing can generally be realized via discontinuous constituent parsing. At best, one can highlight one or two typical implementations of dependency parsing and constituent parsing and argue that the mechanisms for dealing with non-projectivity are comparable in nature, awaiting precise arguments relating their formal power.

One well-established approach to dealing with non-projective dependency structure is commonly referred to as *pseudo-projectivity* (Kahane et al., 1998; Nivre and Nilsson, 2005). The idea is that a first phase of projective parsing is followed by a *lifting* operation that rearranges edges to make them cross one another. A related idea for discontinuous constituent parsing is the reversible splitting conversion of Boyd (2007).

A related but different approach is due to Nivre (2009). Here, the usual one-way input tape is replaced by a buffer. A non-topmost element from the parsing stack, which holds a word previously read from the input sentence, can be transferred back to the buffer, and thereby input positions can be effectively swapped, and non-projective structures result. We see no reason why the same idea would not equally well apply to constituent parsing.

This paper has the following structure. After fixing notation in Section 2, we present a formal model of deterministic parsing in Section 3, in terms of oracle automata. These automata appear at first sight to be biased towards constituent parsing, but Section 4 shows that they allow a clean formalization of arc-standard and arc-eager transition-based dependency parsing. It is shown in Section 5 that oracle automata can, under reasonable assumptions, be transformed into latent-variable context-free grammars. Section 6 further explores these assumptions as relating to common implementations of transition-based dependency parsing. As shown in Section 7, the results carry over to probabilistic automata and grammars.

2 Preliminaries

In this paper, a *tree* refers to a term built of leaf symbols, from the alphabet Σ_{leaf}, and internal symbols, from the alphabet Σ_{intern}. A symbol $a \in \Sigma_{leaf}$ by itself is a tree, and if $A \in \Sigma_{intern}$ and $t_1, \ldots, t_k$ are trees, then $A(t_1 \cdots t_k)$ is a tree. (In this paper, we assume the number k of children of an internal node is non-zero.) The set of all trees with given alphabets Σ_{leaf} and Σ_{intern} is denoted by $\mathcal{T}(\Sigma_{leaf}, \Sigma_{intern})$. We will use the symbol t for trees and symbol τ for sequences of trees. The empty sequence is denoted by ε. We define the *root* of a tree by $root(a) = a$ and $root(A(t_1 \cdots t_k)) = A$. We define the *yield* of a tree by $yield(a) = a$ and $yield(A(t_1 \cdots t_k)) = yield(t_1) \cdots yield(t_k)$. We define $first(aw) = a$ and $last(wa) = a$, for $a \in \Sigma_{leaf}$ and $w \in \Sigma^*_{leaf}$, $first(\varepsilon)$ and $last(\varepsilon)$ are undefined, and $first(t) = first(yield(t))$ and $last(t) = last(yield(t))$ for

$t \in \mathcal{T}(\Sigma_{leaf}, \Sigma_{intern})$.

As usual, a *context-free grammar* (CFG) is represented by a 4-tuple (Σ, N, S, R), where Σ and N are two disjoint finite sets of *terminals* and *nonterminals*, respectively, $S \in N$ is the *start symbol*, and R is a finite set of *rules*, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$. By *grammar symbol* we mean a terminal or nonterminal. We use symbols $A, B, C, \ldots$ for nonterminals, $a, b, c, \ldots$ for terminals, $v, w, x, \ldots$ for strings of terminals, and $\alpha, \beta, \gamma, \ldots$ for strings of grammar symbols. To simplify the discussion we will assume that all rules are of the form $A \rightarrow B_1 \cdots B_k$, where $k \geq 1$, or of the form $A \rightarrow a$. We also assume that S does not occur in the right-hand side of any rule.

A *latent-variable* CFG (L-CFG) differs from a CFG in that each nonterminal, except the start symbol $S^\dagger$, is of the composite form $A^{(\ell)}$, where A is a *surface* symbol and ℓ is a *latent* symbol. We denote the set of latent symbols by L, the set of surface symbols by N and the set of composite nonterminals by N^L. A L-CFG has one or more rules of the form $S^\dagger \rightarrow S^{(\ell)}$, some $\ell \in L$.

The intuition behind L-CFG is that the surface symbols are those that occur in parse trees used to represent syntactic structure, these being trees in $\mathcal{T}(\Sigma, N)$, while the purpose of the latent symbols is to restrict derivations and help define probability distributions over parse trees, once we extend rules with probabilities.

L-CFGs are intimately related to regular tree grammars (Brainerd, 1969; Gécseg and Steinby, 1997), which is apparent from the definition of their 'derives' relation. Fix a L-CFG G. For $t, t' \in \mathcal{T}(\Sigma \cup N^L, N)$, we let $t \Rightarrow_G t'$ if t' results from t by replacing some occurrence of $A^{(\ell)}$ by $A(\alpha)$, for some rule $A^{(\ell)} \rightarrow \alpha$. A *parse tree* of G is a tree $t \in \mathcal{T}(\Sigma, N)$ such that $S^{(\ell)} \Rightarrow_G^* t$, for some ℓ.

A *canonical* L-CFG is formed from a CFG by having a singleton set $L = \{\ell\}$, enhancing each nonterminal occurrence with an additional superscript (ℓ), and adding rule $S^\dagger \rightarrow S^{(\ell)}$. The parse trees of this canonical L-CFG concur with the standard definition of parse tree of a CFG. The set of all parse trees is called the *tree language* of a CFG or L-CFG.

It is often convenient to distinguish a subset of the composite nonterminals as not producing any surface symbols in parse trees. For example, if we binarize a long rule $A \rightarrow B\ C\ D$ from a CFG into two rules $A^{(\ell_A)} \rightarrow E^{(\ell_E)}\ D^{(\ell_D)}$ and $E^{(\ell_E)} \rightarrow B^{(\ell_B)} C^{(\ell_C)}$ of a L-CFG, then we may wish to mark $E^{(\ell_E)}$ as not producing any trace in the parse tree. In terms of the derives relation, this means $A^{(\ell_A)} \Rightarrow_G A(E^{(\ell_E)} D^{(\ell_D)}) \Rightarrow_G A(B^{(\ell_B)} C^{(\ell_C)} D^{(\ell_D)})$, etc. The same principle may be used to avoid spurious ambiguity in the representation of dependency structures using bilexical context-free grammars (Section 4).

3 Oracle automata

We define an *oracle automaton* as a variant of a traditional shift-reduce parser, in which an oracle uniquely determines the next parser action. The oracle is a partial function Ω that maps a sequence of trees (the current content of the *stack*, see below) and a terminal (the *lookahead*) to a rule that is to be applied next. It is constrained by:

1. if $\Omega(\tau, a) = (A \rightarrow b)$, then $a = b$; and
2. if $\Omega(\tau, a) = (A \rightarrow B_1 \cdots B_k)$, then τ can be written as $\tau' t_1 \cdots t_k$, where $root(t_i) = B_i$ for each i $(1 \leq i \leq k)$.

The first constraint says that reduction of terminal b to nonterminal A should only be suggested if that terminal is the lookahead. The second constraint says that reduction by a rule with right-hand side of length k should only be suggested if the roots of the top-most k subtrees on the stack match the right-hand side of that rule.

For CFG G and oracle Ω, the oracle automaton $\mathcal{M}$ manipulates configurations of the form $(\tau, v\$)$, where τ is the stack, and v is the remaining input. The symbol \$ is the end-of-sentence marker, which we will need for technical reasons. For given input w, the initial configuration is $(\varepsilon, w\$)$. The allowable steps are:

shift $(\tau, av\$) \vdash_{\mathcal{M}} (\tau A(a), v\$)$, if $\Omega(\tau, a) = (A \rightarrow a)$; and

reduce $(\tau t_1 \cdots t_k, v\$) \vdash_{\mathcal{M}} (\tau A(t_1 \cdots t_k), v\$)$ if $\Omega(\tau t_1 \cdots t_k, \mathit{first}(v\$)) = (A \rightarrow B_1 \cdots B_k)$.

If a configuration is $(\tau, av\$)$ and $\Omega(\tau, a)$ is undefined, then parsing fails. Acceptance happens upon reaching a configuration $(t, \$)$ with $root(t) = S$. By a *computation* we mean a sequence $(\tau, vw\$) \vdash_{\mathcal{M}}^* (\tau', w\$)$ of zero or more steps.

It is easy to see that if $(\varepsilon, w\$) \vdash^*_{\mathcal{M}} (t, \$)$ then $yield(t) = w$. The set of all t such that $(\varepsilon, w\$) \vdash^*_{\mathcal{M}} (t, \$)$ and $root(t) = S$ is the *tree language* of $\mathcal{M}$, and is a subset of the tree language of G. This may be a *strict* subset. In particular, if t is a parse tree of G and $w = yield(t)$, then there is at most one computation of the form $(\varepsilon, w\$) \vdash^*_{\mathcal{M}} (t', \$)$, for some t' with $root(t') = S$, but t' may or may not be equal to t, depending on the definition of Ω.

In practice, the value $\Omega(\tau, a)$ is not an arbitrary function of τ and a. It will typically depend on only a bounded number of features that can be extracted from τ. As τ becomes longer and as trees in τ become deeper, through application of reductions, more and more details of these trees will be outside the reach of these features. This observation is formalized through the notion of a *stack congruence* $\equiv_{st}$, which is an equivalence relation on stacks, with additional constraints. These additional constraints are to capture the intuition that once details of a stack have become irrelevant to the features, they remain so. It was inspired by Pereira and Wright (1997).

In order to define $\equiv_{st}$, we also need a *tree congruence* $\equiv_{tr}$, which is an equivalence relation on trees, again with an additional constraint. This constraint on $\equiv_{tr}$ says that if $t_1 \equiv_{tr} t'_1$, ..., $t_k \equiv_{tr} t'_k$, then $A(t_1 \cdots t_k) \equiv_{tr} A(t'_1 \cdots t'_k)$ for each A. The intuition is that once we have decided that some aspects of trees $t_1, \ldots t_k$ are no longer useful for the oracle, this remains so when these trees become part of a deeper tree, by adding A as root.

The constraint on $\equiv_{st}$ now says that if $\tau \equiv_{st} \tau'$ and $t \equiv_{tr} t'$ then also $\tau t \equiv_{st} \tau' t'$. We further say that $\equiv_{st}$ is *consistent* with oracle Ω if $\tau \equiv_{st} \tau'$ implies $\Omega(\tau, a) = \Omega(\tau', a)$ for every a. The equivalence class of tree t is denoted by $[t]_{\equiv_{tr}} = \{t' \mid t' \equiv_{tr} t\}$, or simply $[t]$. Similarly, the equivalence class of stack τ is denoted by $[\tau]_{\equiv_{st}}$ or $[\tau]$.

Trivial tree and stack congruences result by equivalence classes that each contain a single tree or stack, respectively. This would entail an infinite number of equivalence classes. As argued above however, we may reasonably assume that the number of equivalence classes is finite, considering typical oracles would use a bounded number of features. These features are likely to investigate only a bounded number of top-most trees on the stack, and for any such tree, the focus of interest is likely to be the root, or leaves at the extreme ends of the yields. Features are further discussed in Section 6.

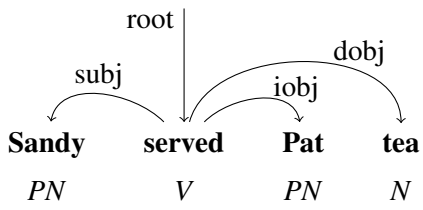

Figure 1: Example dependency structure.

For technical reasons, we will assume $t \equiv_{tr} t'$ implies $first(t) = first(t')$. This is without loss of generality: in the worst case, one needs to split up each existing equivalence class into several, one for each terminal.

4 Oracle automata and transition-based dependency parsing

As our oracle automata were defined in terms of context-free grammars, it deserves an explanation how we can use them to perform dependency parsing. The most straightforward solution is to assume a bilexical context-free grammar, with a single delexicalized nonterminal A. Concretely, for every pair of words a and b, we assume two rules $A_a \to A_a A_b$ and $A_a \to A_b A_a$. In the first rule, the first member in the right-hand side is the *head*, and in the second rule, the second member is the *head*. (The critical reader may object this definition is inconsistent if $a = b$; this could be fixed by having two versions of each nonterminal A_a.) For each terminal, we also have the rule $A_a \to a$ and the rule $S \to A_a$.

With grammars of this form, we obtain what is commonly known as *spurious ambiguity*. That is, there may be several parse trees that correspond to the same dependency structure. For example, the dependency structure in Figure 1 can be obtained by a left-most derivation: $A_{\mathbf{served}} \Rightarrow A_{\mathbf{Sandy}}\ A_{\mathbf{served}} \Rightarrow \mathbf{Sandy}\ A_{\mathbf{served}} \Rightarrow \mathbf{Sandy}\ A_{\mathbf{served}}\ A_{\mathbf{tea}} \Rightarrow \mathbf{Sandy}\ A_{\mathbf{served}}\ A_{\mathbf{Pat}}\ A_{\mathbf{tea}} \Rightarrow^* \mathbf{Sandy\ served\ Pat\ tea}$. Two more left-most derivations exist however that correspond to the same dependency structure, both starting with $A_{\mathbf{served}} \Rightarrow A_{\mathbf{served}}\ A_{\mathbf{tea}}$.

In practice, this spurious ambiguity is not a

problem. It is the oracle that ensures that only one structure is produced. Spurious ambiguity of transition-based dependency parsing is discussed at length by Goldberg and Nivre (2012).

If we apply oracle automata on the above bilexical context-free grammars, we obtain what Nivre et al. (2007) call the *arc-standard* strategy of transition-based dependency parsing. This contrasts with the *arc-eager* strategy. The latter has a *shift* operation, which corresponds exactly to our shift. The *left-arc* operation corresponds to reduction with a rule $A_a \rightarrow A_b A_a$, or in other words, a step $(\tau A_b(\tau_b) A_a(\tau_a), v\$) \vdash (\tau A_a(A_b(\tau_b) A_a(\tau_a)), v\$)$. The formulation of e.g. Nivre et al. (2007) has the top of the stack as part of the remaining input, which is largely an inconsequential notational difference, although it does affect the way features address elements in the stack or in remaining input; we will return to this issue in Section 6.

Contrary to what one may expect, the *right-arc* operation is not the mirror image of the left-arc operation but then for the rule $A_a \rightarrow A_a A_b$. The easiest way of looking at the right-arc operation is as making an early commitment to do the actual *reduce* operation with the rule $A_a \rightarrow A_a A_b$, before all the dependents of b have been processed.

In terms of bilexical grammars, this 'early commitment' made by the right-arc operation can be expressed by marking a nonterminal occurrence, to enforce that it (or its ancestors) will end up as the second member (as opposed to the first) in the right-hand side of a rule. We will use a bar-symbol for this mark. Concretely, we may construct the following rules:

- $S \rightarrow A_a$, $A_a \rightarrow a$ and $\overline{A}_a \rightarrow A_a$, for each a, and
- $A_a \rightarrow A_b A_a$, $A_a \rightarrow A_a \overline{A}_b$ and $\overline{A}_a \rightarrow \overline{A}_a \overline{A}_b$, for each pair a and b.

A reduction with rule $\overline{A}_a \rightarrow A_a$ now corresponds to a right-arc operation, and a reduction with $A_a \rightarrow A_a \overline{A}_b$ or $\overline{A}_a \rightarrow \overline{A}_a \overline{A}_b$ corresponds to what is called a reduce in the arc-eager model.

Having a representation of 'early commitment' by bar-symbols does not change the information available to an oracle, relative to the formulation of the right-arc operation in the cited literature. In the worst case, it will require a different way of addressing elements in the stack. Thereby the conclusions we will draw in Section 6 are unaffected.

5 Construction of a L-CFG from an oracle automaton

Let us assume an oracle automaton $\mathcal{M}$ for a CFG G and an oracle Ω. We also assume a tree congruence $\equiv_{tr}$ and a stack congruence $\equiv_{st}$ consistent with Ω, both with a finite number of equivalence classes. We will construct a L-CFG $G_{\mathcal{M}}$ as follows. The terminals of $G_{\mathcal{M}}$ are those of G. The nonterminals of $G_{\mathcal{M}}$ are S and composite nonterminals of the form $A^{(\ell)}$, where A is a nonterminal from G and the latent symbol ℓ is a triple $([\tau], [t], a)$ consisting of an equivalence class of stacks, an equivalence class of trees, and a lookahead symbol. Intuitively, $[\tau]$ represents context to the left of the occurrence of A, $[t]$ captures internal properties of a derivation of A, and a represents the first terminal of context immediately to the right of the occurrence of A.

There are three types of rules in $G_{\mathcal{M}}$. The first is:

$$S^{\dagger} \rightarrow S^{([\varepsilon],[t],\$)}$$

for every class $[t]$. This is easily justified, as initially the stack is empty, and the first symbol after an occurrence of S must be \$. The second is:

$$A^{([\tau],[A(a)],b)} \rightarrow a$$

for every class $[\tau]$, terminals a and b, and rule $(A \rightarrow a) = \Omega(\tau, a)$. The third is:

$$A^{([\tau_0],[t_0],a_0)} \rightarrow B_1^{([\tau_1],[t_1],a_1)} \cdots B_k^{([\tau_k],[t_k],a_k)}$$

for all classes $[\tau_0]$, $[\tau_1]$, …, $[\tau_k]$, classes $[t_0]$, $[t_1]$, …, $[t_k]$, terminals a_0, a_1, …, a_k, and rule $(A \rightarrow B_1 \cdots B_k) = \Omega(\tau_k, a_0)$ such that:

- $[\tau_1] = [\tau_0]$ and $[\tau_i] = [\tau_{i-1} t_{i-1}]$ for each i $(1 < i \leq k)$,
- $[t_0] = [A(t_1 \cdots t_k)]$,
- $a_i = \mathit{first}(t_{i+1})$ for each i $(1 \leq i < k)$ and $a_k = a_0$.

Note that the definitions are all well-defined. For example, $[\tau_i] = [\tau_{i-1} t_{i-1}]$ uniquely denotes an equivalence class, regardless of the choice of τ_{i-1} from $[\tau_{i-1}]$ and the choice of t_{i-1} from $[t_{i-1}]$, because of $\equiv_{st}$ being a stack congruence. Similarly, $a_i = \mathit{first}(t_{i+1})$ is well-defined by the additional assumption on tree congruences. The above construction is reminiscent of covering of LR(k) grammars by LR(1) grammars (Sippu and Soisalon-Soininen, 1990).

Theorem 1 *The tree language of $G_{\mathcal{M}}$ equals the tree language of $\mathcal{M}$.*

Proof. It is easy to see that if $A^{([\tau],[t'],b)} \Rightarrow^*_{G_{\mathcal{M}}} t$, for some A, t', b and t, then $[t'] = [t]$. We now need to show that $(\varepsilon, w\$) \vdash^*_{\mathcal{M}} (t, \$)$ if and only if $S^{([\varepsilon],[t],\$)} \Rightarrow^*_{G_{\mathcal{M}}} t$, for every $t \in \mathcal{T}(\Sigma, N)$, where $w = \mathit{yield}(t)$.

In the 'only if' direction, it suffices to prove by induction on the length of computations that $(\tau, vw\$) \vdash^*_{\mathcal{M}} (\tau t, w\$)$ implies $A^{([\tau],[t],b)} \Rightarrow^*_{G_{\mathcal{M}}} t$, where $A = \mathit{root}(t)$ and $b = \mathit{first}(w\$)$.

The base case applies if $v = a$ and the computation consists of a shift $(\tau, aw\$) \vdash_{\mathcal{M}} (\tau A(a), w\$)$, where $\Omega(\tau, a) = (A \to a)$. Then $G_{\mathcal{M}}$ must include a rule $A^{([\tau],[A(a)],b)} \to a$, where $b = \mathit{first}(w\$)$. Hence $A^{([\tau],[A(a)],b)} \Rightarrow_{G_{\mathcal{M}}} A(a)$.

Otherwise, we have a computation:

$$
\begin{aligned}
&(\tau_0, v_1 \cdots v_k w\$) \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1, v_2 \cdots v_k w\$) \vdash^*_{\mathcal{M}} \cdots \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1 \cdots t_{k-1}, v_k w\$) \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1 \cdots t_k, w\$) \vdash_{\mathcal{M}} (\tau_0 A(t_1 \cdots t_k), w\$)
\end{aligned}
$$

where $\Omega(\tau_0 t_1 \cdots t_k, a_0) = (A \to B_1 \cdots B_k)$, with $a_0 = \mathit{first}(w\$)$ and $B_i = \mathit{root}(t_i)$ for each i ($1 \leq i \leq k$).

Let further $\tau_1 = \tau_0$ and $\tau_i = \tau_{i-1} t_{i-1}$ for each i ($1 < i \leq k$), let $t_0 = A(t_1 \cdots t_k)$, let $a_i = \mathit{first}(t_{i+1})$ for each i ($1 \leq i < k$) and let $a_k = a_0$. Then $G_{\mathcal{M}}$ must include a rule:

$$A^{([\tau_0],[t_0],a_0)} \to B_1^{([\tau_1],[t_1],a_1)} \cdots B_k^{([\tau_k],[t_k],a_k)}$$

We can now use the inductive hypothesis, which tells us that $B_i^{([\tau_i],[t_i],a_i)} \Rightarrow^*_{G_{\mathcal{M}}} t_i$ for each i ($1 \leq i \leq k$). Hence $A^{([\tau_0],[t_0],a_0)} \Rightarrow^*_{G_{\mathcal{M}}} t_0$.

In the 'if' direction, it suffices to prove by induction on the tree depth that $A^{([\tau],[t],b)} \Rightarrow^*_{G_{\mathcal{M}}} t$ implies $(\tau, vw\$) \vdash^*_{\mathcal{M}} (\tau t, w\$)$ for every w with $b = \mathit{first}(w\$)$, where $v = \mathit{yield}(t)$.

The base case applies if the derivation is $A^{([\tau],[A(a)],b)} \Rightarrow_{G_{\mathcal{M}}} A(a)$. Due to existence of $A^{([\tau],[A(a)],b)} \to a$, and because $\equiv_{st}$ is consistent with Ω, we must have $\Omega(\tau, a) = (A \to a)$. Hence $(\tau, aw\$) \vdash_{\mathcal{M}} (\tau A(a), w\$)$ for every w, regardless of whether $b = \mathit{first}(w\$)$.

Otherwise, we have a derivation:

$$
\begin{aligned}
&A^{([\tau_0],[t_0],a_0)} \Rightarrow_{G_{\mathcal{M}}} \\
&A(B_1^{([\tau_1],[t_1],a_1)} \cdots B_k^{([\tau_k],[t_k],a_k)}) \Rightarrow^*_{G_{\mathcal{M}}} \\
&A(t_1 \cdots t_k)
\end{aligned}
$$

for some stack τ_0, trees $t_1, \ldots, t_k$, and terminal a_0 such that $\Omega(\tau_k, a_0) = (A \to B_1 \cdots B_k)$, where $\tau_1 = \tau_0$, $\tau_i = \tau_{i-1} t_{i-1}$ for each i ($1 < i \leq k$), $t_0 = A(t_1 \cdots t_k)$, $a_i = \mathit{first}(t_{i+1})$ for each i ($1 \leq i < k$) and $a_k = a_0$.

Let $v_i = \mathit{yield}(t_i)$ for each i ($1 \leq i \leq k$). Choose w such that $\mathit{first}(w\$) = a_0$. This means $a_i = \mathit{first}(v_{i+1} \cdots v_k w\$)$ for each i ($1 \leq i \leq k$). We can now apply the inductive hypothesis on $B_i^{([\tau_i],[t_i],a_i)} \Rightarrow^*_{G_{\mathcal{M}}} t_i$ for each i ($1 \leq i \leq k$), and assemble the desired computation:

$$
\begin{aligned}
&(\tau_0, v_1 \cdots v_k w\$) \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1, v_2 \cdots v_k w\$) \vdash^*_{\mathcal{M}} \cdots \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1 \cdots t_{k-1}, v_k w\$) \vdash^*_{\mathcal{M}} \\
&(\tau_0 t_1 \cdots t_k, w\$) \vdash_{\mathcal{M}} (\tau_0 A(t_1 \cdots t_k), w\$)
\end{aligned}
$$

Note that we made implicit use of $\equiv_{st}$ being consistent with Ω, so that, for example, the choice of τ_0 from a class $[\tau_0]$ is irrelevant. ∎

Our construction may result in a L-CFG that is not reduced, that is, it may contain unreachable or unproductive rules. This can be solved by reduction algorithms for CFGs (Harrison, 1978).

If probabilistic L-CFGs are desired, one may assign probabilities in an arbitrary way, for example by assigning probability $1/n$ to each rule that shares its left-hand side with $n - 1$ other rules. This does not change the tree language however, and the grammar remains unambiguous, that is, for each string, there is at most one tree.

We make no claim that the construction of L-CFGs as given here has practical benefits over methods for obtaining L-PCFGs via EM training or spectral learning. The main purpose of our construction was to show that L-(P)CFGs are at least as powerful as oracle automata.

6 Features

We now present a formalization of common features. Recall *root* as defined in Section 2. We introduce $\perp$ to denote the undefined value. We assume a function applied on the undefined value evaluates to the undefined value; for example $\mathit{root}(\perp) = \perp$.

We define *child* and *nth* by $\mathit{child}(i, A(t_1 \cdots t_k)) = \mathit{child}(-i, A(t_k \cdots t_1)) = \mathit{nth}(i, t_1 \cdots t_k) = \mathit{nth}(-i, t_k \cdots t_1) = t_i$, for $1 \leq i \leq k$. In words, the first argument is an index, which counts from the left if it is positive and from the right if it is negative. For arguments

not covered by the above, the function values are $\perp$.

We now define a *feature* to be a function F from sequences of trees (stacks) to $\Sigma_{leaf} \cup \Sigma_{intern} \cup \{\perp\}$. We will first consider a simple kind of feature of the form $F = root(child(i_\ell, child(i_{\ell-1}, \ldots, child(i_1, nth(i_0, \cdot)\ldots)$, some $\ell \geq 0$. Here $\cdot$ is a placeholder for the stack as argument. In words, the feature value for a stack $t_1 \cdots t_k$ is found by considering t_{i_0} if $i_0 > 0$ or t_{k+1+i_0} if $i_0 < 0$. The subtree at index i_1 is then taken (distinguishing between $i_1 > 0$ and $i_1 < 0$ as before), etc. Of the subtree obtained by the final application of $child$ with argument i_ℓ the root label is returned.

For F as above we define $initial(F) = i_0$ and for $0 \leq j \leq \ell$, we let $prefix(F, j)$ denote the function $root(child(i_\ell, child(i_{\ell-1}, \ldots, child(i_{j+1}, \cdot)\ldots)$. In words, from F we remove the initial application of nth and the next j applications of $child$. We let $prefixes(F) = \{prefix(F, j) \mid 0 \leq j \leq \ell\}$. For a function of the form $prefix(F, j)$, we let $head(prefix(F, j)) = i_{j+1}$ and $tail(prefix(F, j)) = prefix(F, j + 1)$ for $0 \leq j < \ell$, and $head(prefix(F, \ell)) = tail(prefix(F, \ell)) = \perp$. In words, $head$ returns the index of the next application of $child$ if there is one, and $tail$ removes the next application of $child$.

We say oracle Ω is determined by the sequence $F_1, \ldots, F_f$ of features if for every τ_1, τ_2, a_1, a_2, the equalities $F_j(\tau_1) = F_j(\tau_2)$ for every j ($1 \leq j \leq f$) and $a_1 = a_2$ together imply $\Omega(\tau_1, a_1) = \Omega(\tau_2, a_2)$. Here we have treated the lookahead (a_1 or a_2) as an implicit feature.

In order to obtain a tree congruence from a sequence $F_1, \ldots, F_f$ of simple features as above, we first define $\mathcal{F} = \cup_{1 \leq j \leq f} prefixes(F_j)$. Next we define a function $erase$, which erases from a tree t all subtrees that are outside the reach of the functions in $\mathcal{F}$, and that remain so if t becomes a subtree of a bigger tree. Erasing is done by removing subtrees or replacing them by $\perp$. Formally, for $\mathcal{G} \subseteq \mathcal{F}$, $erase(\mathcal{G}, t) = \perp$ if $\mathcal{G} = \emptyset$, and for $\mathcal{G} \neq \emptyset$ we define:

$$erase(\mathcal{G}, A(t_1 \cdots t_k)) \quad = \quad A(t'_1 \cdots t'_{k'} t''_{k''} \cdots t''_k)$$

where the t'_i and t''_i are defined below. First, let $i_{max} = \max\{i \in head(F) \mid F \in \mathcal{G}, 1 \leq i \leq k\}$ and $i_{min} = \min\{i \in head(F) \mid F \in \mathcal{G}, 1 \leq -i \leq k\}$. In words, in potential next applications of $child$ in functions in $\mathcal{G}$, we consider the indices counting from the left and those counting from the right and take the rightmost and leftmost, respectively of those indices. If i_{max} is defined then $k' = i_{max}$ and otherwise $k' = 0$. If i_{min} is defined then $k'' = k + 1 + i_{min}$ and otherwise $k'' = k + 1$. Note that k' may be greater than $k'' - 1$.

For $1 \leq i \leq k'$ we now define $t'_i = erase(\mathcal{G}'_i, t_i)$ where $\mathcal{G}'_i = \{tail(F) \mid F \in \mathcal{G}, head(F) = i\}$. For $k'' \leq i \leq k$ we define $t''_i = erase(\mathcal{G}''_i, t_i)$ where $\mathcal{G}''_i = \{tail(F) \mid F \in \mathcal{G}, k + 1 + head(F) = i\}$. Note that the total number of nodes in (the tree representations of) the functions in $\mathcal{G}'_i$ and $\mathcal{G}''_i$ is strictly smaller than the total number of nodes in the trees in $\mathcal{G}$. It follows that, for any t, the size of tree $erase(\mathcal{F}, t)$ is bounded, that is, the set of such trees is finite.

Define $\equiv_{tr}$ by $t_1 \equiv_{tr} t_2$ if and only if $erase(\mathcal{F}, t_1) = erase(\mathcal{F}, t_2)$. By the definition of $erase$, we have $erase(\mathcal{F}, A(t_1 \cdots t_k)) = erase(\mathcal{F}, A(erase(\mathcal{F}, t_1) \cdots erase(\mathcal{F}, t_k)))$ for every tree $A(t_1 \cdots t_k)$. It follows that $\equiv_{tr}$ is a (finite) tree congruence.

As an example, consider $f = 1$ and:

$$F_1 = root(child(3, child(2, nth(-1, \cdot))))$$
$$t = A(B_1(b_1)B_2(C_1(c_1)C_2(c_2)C_3(c_3))B_3(b_3))$$

Then $\mathcal{F} = \{root(\cdot), root(child(3, \cdot)), root(child(3, child(2, \cdot)))\}$ and $erase(\mathcal{F}, t) = A(\perp B_2(\perp\perp C_3())B_3())$.

In order to obtain our (finite) stack congruence $\equiv_{st}$, we erase elements from a stack $t_1 \cdots t_k$. We now determine $i_{max} = \max\{initial(F_j) \mid 1 \leq j \leq f, 1 \leq initial(F_j) \leq k\}$ and $i_{min} = \min\{initial(F_j) \mid 1 \leq j \leq f, 1 \leq -initial(F_j) \leq k\}$. Much as before we have $k' = i_{max}$ if i_{max} is defined and $k' = 0$ otherwise and $k'' = k + 1 + i_{min}$ if i_{min} is defined and $k'' = k + 1$ otherwise. The stack after erasure is $erase(\mathcal{F}, t_1) \cdots erase(\mathcal{F}, t_{k'}) erase(\mathcal{F}, t_{k''}) \cdots erase(\mathcal{F}, t_k)$. This allows definition of $\equiv_{st}$, in the same way as of $\equiv_{tr}$.

We can extend the repertoire of functions in our features. For example, we can include *first* and *last* as defined in Section 2. We can also add the function *first_intern*, which returns the internal symbol just above the leftmost leaf. Formally, $first\_intern(A(a)) = A$ and $first\_intern(A(t_1 \cdots t_k)) = first\_intern(t_1)$ if $t_1 \notin \Sigma_{leaf}$. The definition of $last\_intern$ is symmetric. Allowing such functions requires appro-

priate refinements of $erase$, such that the depth of the resulting trees remains bounded, by keeping only the relevant nodes near selected leaves.

We will now discuss the features used by the MaltParser, one of the most widely publicized transition-based dependency parsers. The descriptions will be based on Nivre et al. (2006) and Nivre et al. (2007).

All features are defined in terms of word form, part of speech or dependency relation. In our oracle automata, this information can all be encoded as parts of names of terminals and nonterminals. In the MaltParser, the word forms, parts of speech and dependency relations are attached to 'tokens' in the state of the parser. These tokens are found in the stack or in the remaining input.

Tokens can be addressed by an index, which for the stack counts from the top downward (cf. our function nth with negative first argument), and for the remaining input counts rightward from the first unconsumed token. One source of confusion with our automata is that in the MaltParser dependency links can be attached to the first token in the remaining input, whereas our automata would first have to transfer such a token to the stack before linking it to other tokens by means of a reduction. There can therefore be a slight mismatch in the type of addressing of tokens, relative to our formal framework above.

For presentational reasons, we have limited the size of the lookahead of our oracle automata to 1. Without causing any further complications however, this can be relaxed to lookahead of any fixed size. In this way, we can model features of the MaltParser that look a fixed distance ahead in the remaining input. As for the construction of the L-CFG, this would be modified accordingly, with latent symbols in which the third component is a string of the appropriate length.

Next to addressing tokens by index, features of the MaltParser can also refer to leftmost and rightmost dependents of indexed tokens. In our framework, such features could be expressed using functions similar to $child$, $first\_intern$ and $last\_intern$, all allowing $erase$ to return trees of bounded size as before.

Features similar to those of the MaltParser are used by Sagae and Lavie (2005), but in addition, their features also include e.g. the *number* of dependents of a token. This might suggest the number of equivalence classes is infinite after all. In practice however, the oracle would only deal with one from a bounded number of possible values, that is, those that were encountered during training, which is necessarily finite. It is not clear to us how their parser would behave if a value is encountered during testing that is larger than the maximum one encountered during training.

7 The probabilistic case

One may redefine Ω to be a probability distribution, constrained by:

- if $\Omega(A \to b \mid [\tau], a) > 0$, then $a = b$; and
- if $\Omega(A \to B_1 \cdots B_k \mid [\tau], a) > 0$, then τ can be written as $\tau' t_1 \cdots t_k$, where $root(t_i) = B_i$ for each i $(1 \leq i \leq k)$.

This is in the same spirit as the non-deterministic oracles of Goldberg and Nivre (2013).

With Ω now being a probability distribution, we can refine the semantics of our automata to assign a probability to each computation, which is the product of the probabilities of all used steps. The construction from Section 5 can be extended to produce a L-PCFG, where:

- $S^{\dagger} \to S^{([\varepsilon],[t],\$)}$ is assigned probability 1,
- $A^{([\tau],[A(a)],b)} \to a$ is assigned $\Omega(A \to a \mid [\tau], a)$,
- $A^{([\tau_0],[t_0],a_0)} \to B_1^{([\tau_1],[t_1],a_1)} \cdots B_k^{([\tau_k],[t_k],a_k)}$ is assigned $\Omega(A \to B_1 \cdots B_k \mid [\tau_k], a_0)$.

If desired, the L-PCFG can be normalized to become proper, i.e. so that the probabilities of all rules with given left-hand side sum to 1; see e.g. Chi (1999).

8 Conclusions

We have explored formal properties of transition-based dependency parsing, in terms of traditional automata theory. Through our formalization, an explicit link has been established between projective transition-based dependency parsing and constituent parsing, in particular latent-variable context-free parsing. Extension to the non-projective/discontinuous case will be the subject of future investigations.

Acknowledgements

Thanks go to reviewers for helpful comments.

References

A. Boyd. 2007. Discontinuity revisited: An improved conversion to context-free representations. In *Proceedings of the Linguistic Annotation Workshop, at ACL 2007*, pages 41–44, Prague, Czech Republic, June.

W.S. Brainerd. 1969. Tree generating regular systems. *Information and Control*, 14:217–231.

M. Candito, J. Nivre, P. Denis, and E. Henestroza Anguiano. 2010. Benchmarking of statistical dependency parsers for French. In *The 23rd International Conference on Computational Linguistics*, pages 108–116, Beijing, China, August.

S.A. Caraballo and E. Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *Computational Linguistics*, 24(2):275–298.

G. Carroll and E. Charniak. 1992. Two experiments on learning probabilistic dependency grammars from corpora. In *Statistically-Based NLP Techniques, Papers from the AAAI Workshop*, pages 1–13, San Jose.

D. Cer, M.-C. de Marneffe, D. Jurafsky, and C. Manning. 2010. Parsing to Stanford dependencies: Trade-offs between speed and accuracy. In *LREC 2010: Seventh International Conference on Language Resources and Evaluation, Proceedings*, pages 1628–1632, Valletta , Malta, May.

Z. Chi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160.

M. Collins. 2003. Head-driven statistical models for natural language parsing. *Computational Linguistics*, 29(4):589–637.

M.A. Covington. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

J. Eisner and G. Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 457–464, Maryland, USA, June.

H. Gaifman. 1965. Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337.

F. Gécseg and M. Steinby. 1997. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 3*, chapter 1, pages 1–68. Springer, Berlin.

Y. Goldberg and J. Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *The 24th International Conference on Computational Linguistics*, pages 959–976, Mumbai, India, December.

Y. Goldberg and J. Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414.

J. Goodman. 1997. Global thresholding and multiple-pass parsing. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 11–25, Providence, Rhode Island, USA, August.

K. Hall and V. Novák. 2005. Corrective modeling for non-projective dependency parsing. In *Proceedings of the Ninth International Workshop on Parsing Technologies*, pages 42–52, Vancouver, British Columbia, Canada, October.

M.A. Harrison. 1978. *Introduction to Formal Language Theory*. Addison-Wesley.

D.G. Hays. 1964. Dependency theory: A formalism and some observations. *Language*, 40(4):511–525.

L. Huang and K. Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086, Uppsala, Sweden, July.

M. Johnson. 2007. Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with Unfold-Fold. In *45th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 168–175, Prague, Czech Republic, June.

S. Kahane, A. Nasr, and O. Rambow. 1998. Pseudo-projectivity, a polynomially parsable non-projective dependency grammar. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, volume 1, pages 646–652, Montreal, Quebec, Canada, August.

K. Kallmeyer and M. Kuhlmann. 2012. A formal model for plausible dependencies in lexicalized tree adjoining grammar. In *Eleventh International Workshop on Tree Adjoining Grammar and Related Formalisms*, pages 108–116.

L. Kallmeyer and W. Maier. 2010. Data-driven parsing with probabilistic linear context-free rewriting systems. In *The 23rd International Conference on Computational Linguistics*, pages 537–545, Beijing, China, August.

T. Kalt. 2004. Induction of greedy controllers for deterministic treebank parsers. In *Conference on Empirical Methods in Natural Language Processing*, pages 17–24, Barcelona, Spain, July.

D. Klein and C. Manning. 2004. Corpus-based induction of syntactic structure: Models of dependency and constituency. In *42nd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 478–485, Barcelona, Spain, July.

M. Kuhlmann, C. Gómez-Rodríguez, and G. Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *49th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 673–682, Portland, Oregon, June.

M. Kuhlmann. 2013. Mildly non-projective dependency grammar. *Computational Linguistics*, 39(2):355–387.

A. Lavie and M. Tomita. 1993. GLR* – an efficient noise-skipping parsing algorithm for context free grammars. In *Third International Workshop on Parsing Technologies*, pages 123–134, Tilburg (The Netherlands) and Durbuy (Belgium), August.

D. Lin. 1998. A dependency-based method for evaluating broad-coverage parsers. *Natural Language Engineering*, 4(2):97–114.

X. Ma, X. Zhang, H. Zhao, and B.-L. Lu. 2010. Dependency parser for Chinese constituent parsing. In *CIPS-SIGHAN Joint Conference on Chinese Language Processing*.

T. Matsuzaki, Y. Miyao, and J. Tsujii. 2005. Probabilistic CFG with latent annotations. In *43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 75–82, Ann Arbor, Michigan, June.

Y. Miyao, R. Sætre K. Sagae, T. Matsuzaki, and J. Tsujii. 2008. Task-oriented evaluation of syntactic parsers and their representations. In *46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 46–54, Columbus, Ohio, June.

S. Narayan and S.B. Cohen. 2015. Diversity in spectral learning for natural language parsing. In *Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1868–1878, Lisbon, Portugal, September.

M.-J. Nederhof and M. McCaffery. 2014. Deterministic parsing using PCFGs. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 338–347, Gothenburg, Sweden.

J. Nivre and J. Nilsson. 2005. Pseudo-projective dependency parsing. In *43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 99–106, Ann Arbor, Michigan, June.

J. Nivre and M. Scholz. 2004. Deterministic dependency parsing of English text. In *The 20th International Conference on Computational Linguistics*, volume 1, pages 64–70, Geneva, Switzerland, August.

J. Nivre, J. Hall, and J. Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the Eighth Conference on Computational Natural Language Learning*, pages 49–56, Boston, Massachusetts, May.

J. Nivre, J. Hall, and J. Nilsson. 2006. MaltParser: A data-driven parser-generator for dependency parsing. In *LREC 2006: Fifth International Conference on Language Resources and Evaluation, Proceedings*, pages 2216–2219, Genoa, Italy, May.

J. Nivre, J. Hall, J. Nilsson, A. Chanev, G. Eryiğit, S. Kübler, S. Marinov, and E. Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135.

J. Nivre. 2003. An efficient algorithm for projective dependency parsing. In *8th International Workshop on Parsing Technologies*, pages 149–160, LORIA, Nancy, France, April.

J. Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore, August.

F.C.N. Pereira and R.N. Wright. 1997. Finite-state approximation of phrase-structure grammars. In E. Roche and Y. Schabes, editors, *Finite-State Language Processing*, pages 149–173. MIT Press.

S. Petrov, L. Barrett, R. Thibaux, and D. Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia, July.

O. Rambow. 2010. The simple truth about dependency and phrase structure representations: An opinion piece. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Proceedings of the Main Conference*, pages 337–340, Los Angeles, California, June.

A. Ratnaparkhi. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing*, pages 1–10, Providence, Rhode Island, USA, August.

K. Sagae and A. Lavie. 2005. A classifier-based parser with linear run-time complexity. In *Proceedings of the Ninth International Workshop on Parsing Technologies*, pages 125–132, Vancouver, British Columbia, Canada, October.

S. Seneff. 1989. TINA: A probabilistic syntactic parser for speech understanding systems. In *ICASSP-89*, volume 2, pages 711–714, Glasgow.

S. Sippu and E. Soisalon-Soininen. 1990. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.

Y. Tsuruoka and J. Tsujii. 2005. Chunk parsing revisited. In *Proceedings of the Ninth International Workshop on Parsing Technologies*, pages 133–140, Vancouver, British Columbia, Canada, October.

A. van Cranenburgh, R. Scha, and F. Sangati. 2011. Discontinuous data-oriented parsing: A mildly context-sensitive all-fragments grammar. In *Proceedings of the Second Workshop on Statistical Parsing of Morphologically Rich Languages*, pages 34–44, Dublin, Ireland.

A. Wong and D. Wu. 1999. Learning a lightweight robust deterministic parser. In *Sixth European Conference on Speech Communication and Technology*, pages 2047–2050.

H. Yamada and Y. Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *8th International Workshop on Parsing Technologies*, pages 195–206, LORIA, Nancy, France, April.