# Constrained Arc-Eager Dependency Parsing

Joakim Nivre*
Uppsala University

Yoav Goldberg**
Bar-Ilan University

Ryan McDonald†
Google



*Arc-eager dependency parsers process sentences in a single left-to-right pass over the input and have linear time complexity with greedy decoding or beam search. We show how such parsers can be constrained to respect two different types of conditions on the output dependency graph: span constraints, which require certain spans to correspond to subtrees of the graph, and arc constraints, which require certain arcs to be present in the graph. The constraints are incorporated into the arc-eager transition system as a set of preconditions for each transition and preserve the linear time complexity of the parser.*

## 1. Introduction

Data-driven dependency parsers in general achieve high parsing accuracy without relying on hard constraints to rule out (or prescribe) certain syntactic structures (Yamada and Matsumoto 2003; Nivre, Hall, and Nilsson 2004; McDonald, Crammer, and Pereira 2005; Zhang and Clark 2008; Koo and Collins 2010). Nevertheless, there are situations where additional information sources, not available at the time of training the parser, may be used to derive hard constraints at parsing time. For example, Figure 1 shows the parse of a greedy arc-eager dependency parser trained on the *Wall Street Journal* section of the Penn Treebank before (left) and after (right) being constrained to build a single subtree over the span corresponding to the named entity "Cat on a Hot Tin Roof," which does not occur in the training set but can easily be found in on-line databases. In this case, adding the span constraint fixes both prepositional phrase attachment errors. Similar constraints can also be derived from dates, times, or other measurements that can often be identified with high precision using regular expressions (Karttunen et al. 1996), but are under-represented in treebanks.

* Uppsala University, Department of Linguistics and Philology, Box 635, SE-75126, Uppsala, Sweden.
E-mail: joakim.nivre@lingfil.uu.se.
** Bar-Ilan University, Department of Computer Science, Ramat-Gan, 5290002, Israel.
E-mail: yoav.goldberg@gmail.com.
† Google, 76 Buckingham Palace Road, London SW1W9TQ, United Kingdom.
E-mail: ryanmcd@google.com.

Submission received: 26 June 2013; accepted for publication: 10 October 2013.

doi:10.1162/COLI_a_00184

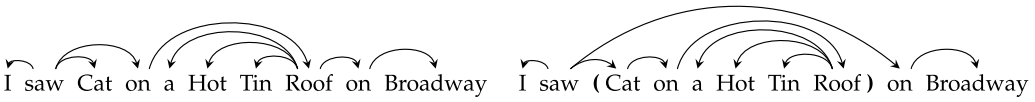


**Figure 1**
Span constraint derived from a title assisting parsing. Left: unconstrained. Right: constrained.

In this article, we examine the problem of constraining transition-based dependency parsers based on the arc-eager transition system (Nivre 2003, 2008), which perform a single left-to-right pass over the input, eagerly adding dependency arcs at the earliest possible opportunity, resulting in linear time parsing. We consider two types of constraints: **span constraints**, exemplified earlier, require the output graph to have a single subtree over one or more (non-overlapping) spans of the input; **arc constraints** instead require specific arcs to be present in the output dependency graph. The main contribution of the article is to show that both span and arc constraints can be implemented as efficiently computed preconditions on parser transitions, thus maintaining the linear runtime complexity of the parser.[1]

Demonstrating accuracy improvements due to hard constraints is challenging, because the phenomena we wish to integrate as hard constraints are by definition not available in the parser's training and test data. Moreover, adding hard constraints may be desirable even if it does not improve parsing accuracy. For example, many organizations have domain-specific gazetteers and want the parser output to be consistent with these even if the output disagrees with gold treebank annotations, sometimes because of expectations of downstream modules in a pipeline. In this article, we concentrate on the theoretical side of constrained parsing, but we nevertheless provide some experimental evidence illustrating how hard constraints can improve parsing accuracy.

## 2. Preliminaries and Notation

*Dependency Graphs.* Given a set $L$ of dependency labels, we define a **dependency graph** for a sentence $x = w_1, \ldots, w_n$ as a labeled directed graph $G = (V_x, A)$, consisting of a set of nodes $V_x = \{1, \ldots, n\}$, where each node $i$ corresponds to the linear position of a word $w_i$ in the sentence, and a set of labeled arcs $A \subseteq V_x \times L \times V_x$, where each arc $(i, l, j)$ represents a dependency with head $w_i$, dependent $w_j$, and label $l$. We assume that the final word $w_n$ is always a dummy word ROOT and that the corresponding node $n$ is a designated root node.

Given a dependency graph $G$ for sentence $x$, we say that a subgraph $G_{[i,j]} = (V_{[i,j]}, A_{[i,j]})$ of $G$ is a **projective spanning tree** over the interval $[i, j]$ $(1 \leq i \leq j \leq n)$ iff (i) $G_{[i,j]}$ contains all nodes corresponding to words between $w_i$ and $w_j$ inclusive, (ii) $G_{[i,j]}$ is a directed tree, and (iii) it holds for every arc $(i, l, j) \in G_{[i,j]}$ that there is a directed path

1 Although span and arc constraints can easily be added to other dependency parsing frameworks, this often affects parsing complexity. For example, in graph-based parsing (McDonald, Crammer, and Pereira 2005) arc constraints can be enforced within the $O(n^3)$ Eisner algorithm (Eisner 1996) by pruning out inconsistent chart cells, but span constraints require the parser to keep track of full subtree end points, which would necessitate the use of $O(n^4)$ algorithms (Eisner and Satta 1999).

from $i$ to every node $k$ such that $\min(i,j) < k < \max(i,j)$ (projectivity). We now define two constraints on a dependency graph $G$ for a sentence $x$:

- $G$ is a **projective dependency tree** (PDT) if and only if it is a projective spanning tree over the interval $[1,n]$ rooted at node $n$.
- $G$ is a **projective dependency graph** (PDG) if and only if it can be extended to a projective dependency tree simply by adding arcs.

It is clear from the definitions that every PDT is also a PDG, but not the other way around. Every PDG can be created by starting with a PDT and removing some arcs.

*Arc-Eager Transition-Based Parsing.* In the arc-eager transition system of Nivre (2003), a **parser configuration** is a triple $c = (\Sigma|i,j|\beta,A)$ such that $\Sigma$ and $B$ are disjoint sublists of the nodes $V_x$ of some sentence $x$, and $A$ is a set of dependency arcs over $V_x$ (and some label set $L$). Following Ballesteros and Nivre (2013), we take the initial configuration for a sentence $x = w_1,\ldots,w_n$ to be $c_s(x) = ([\,],[1,\ldots,n],\{\,\})$, where $n$ is the designated root node, and we take a terminal configuration to be any configuration of the form $c = ([\,],[n],A)$ (for any arc set $A$). We will refer to the list $\Sigma$ as the **stack** and the list $B$ as the **buffer**, and we will use the variables $\sigma$ and $\beta$ for arbitrary sublists of $\Sigma$ and $B$, respectively. For reasons of perspicuity, we will write $\Sigma$ with its head (top) to the right and $B$ with its head to the left. Thus, $c = (\sigma|i,j|\beta,A)$ is a configuration with the node $i$ on top of the stack $\Sigma$ and the node $j$ as the first node in the buffer $B$.

There are four types of **transitions** for going from one configuration to the next, defined formally in Figure 2 (disregarding for now the Added Preconditions column):

- LEFT-ARC$_l$ adds the arc $(j,l,i)$ to $A$, where $i$ is the node on top of the stack and $j$ is the first node in the buffer, and pops the stack. It has as a precondition that the token $i$ does not already have a head.
- RIGHT-ARC$_l$ adds the arc $(i,l,j)$ to $A$, where $i$ is the node on top of the stack and $j$ is the first node in the buffer, and pushes $j$ onto the stack. It has as a precondition that $j \neq n$.
- REDUCE pops the stack and requires that the top token has a head.
- SHIFT removes the first node in the buffer and pushes it onto the stack, with the precondition that $j \neq n$.

A **transition sequence** for a sentence $x$ is a sequence $C_{0,m} = (c_0,c_1,\ldots,c_m)$ of configurations, such that $c_0$ is the initial configuration $c_s(x)$, $c_m$ is a terminal configuration, and there is a legal transition $t$ such that $c_i = t(c_{i-1})$ for every $i$, $1 \leq i \leq m$. The dependency graph derived by $C_{0,m}$ is $G_{c_m} = (V_x, A_{c_m})$, where $A_{c_m}$ is the set of arcs in $c_m$.

*Complexity and Correctness.* For a sentence of length $n$, the number of transitions in the arc-eager system is bounded by $2n$ (Nivre 2008). This means that a parser using greedy inference (or constant width beam search) will run in $O(n)$ time provided that transitions plus required precondition checks can be performed in $O(1)$ time. This holds for the arc-eager system and, as we will demonstrate, its constrained variants as well.

The arc-eager transition system as presented here is sound and complete for the set of PDTs (Nivre 2008). For a specific sentence $x = w_1,\ldots,w_n$, this means that any transition sequence for $x$ produces a PDT (soundness), and that any PDT for $x$ is generated by

| Transition | | Added Preconditions |
|---|---|---|
| LEFT-ARC$_l$ | $(\sigma\|i,j\|\beta,A) \Rightarrow (\sigma,j\|\beta,A\cup\{(j,l,i)\})$ | ARC CONSTRAINTS |
| | | $\neg\exists a \in A_C : d_a = i \wedge [h_a \in \beta \vee l_a \neq l]$ |
| | $\neg\exists a \in A : d_a = i$ | $\neg\exists a \in A_C : h_a = i \wedge d_a \in j\|\beta$ |
| | | SPAN CONSTRAINTS |
| | | $\neg$[IN-SPAN($i$) $\wedge$ $s(i) = s(j) \wedge i = r(s(i))$] |
| | | $\neg$[IN-SPAN($i$) $\wedge$ $s(i) \neq s(j) \wedge i \neq r(s(i))$] |
| | | $\neg$[NONE $\wedge$ IN-SPAN($j$) $\wedge$ $s(i) \neq s(j)$] |
| | | $\neg$[ROOT $\wedge$ IN-SPAN($j$) $\wedge$ $s(i) \neq s(j) \wedge j \neq r(s(j))$] |
| RIGHT-ARC$_l$ | $(\sigma\|i,j\|\beta,A) \Rightarrow (\sigma\|i\|j,\beta,A\cup\{(i,l,j)\})$ | ARC CONSTRAINTS |
| | | $\neg\exists a \in A_C : d_a = j \wedge [h_a \in \sigma \vee l_a \neq l]$ |
| | $j \neq n$ | $\neg\exists a \in A_C : h_a = j \wedge d_a \in i\|\sigma$ |
| | | SPAN CONSTRAINTS |
| | | $\neg$[ENDS-SPAN($j$) $\wedge$ #CC $> 1$] |
| | | $\neg$[IN-SPAN($j$) $\wedge$ $s(i) = s(j) \wedge j = r(s(j))$] |
| | | $\neg$[IN-SPAN($j$) $\wedge$ $s(i) \neq s(j) \wedge j \neq r(s(j))$] |
| | | $\neg$[NONE $\wedge$ IN-SPAN($i$) $\wedge$ $s(i) \neq s(j)$] |
| | | $\neg$[ROOT $\wedge$ IN-SPAN($i$) $\wedge$ $s(i) \neq s(j) \wedge i \neq r(s(i))$] |
| REDUCE | $(\sigma\|i,j\|\beta,A) \Rightarrow (\sigma,j\|\beta,A)$ | ARC CONSTRAINTS |
| | | $\neg\exists a \in A_C : h_a = i \wedge d_a \in j\|\beta$ |
| | $\exists a \in A : d_a = i$ | SPAN CONSTRAINTS |
| | | $\neg$[IN-SPAN($i$) $\wedge$ $s(i) = s(j) \wedge i = r(s(i))$] |
| SHIFT | $(\sigma,i\|\beta,A) \Rightarrow (\sigma\|i,\beta,A)$ | ARC CONSTRAINTS |
| | | $\neg\exists a \in A_C : d_a = j \wedge h_a \in i\|\sigma$ |
| | $i \neq n$ | $\neg\exists a \in A_C : h_a = j \wedge d_a \in i\|\sigma$ |
| | | SPAN CONSTRAINTS |
| | | $\neg$[ENDS-SPAN($j$) $\wedge$ #CC $> 0$] |

**Figure 2**
Transitions for the arc-eager transition system with preconditions for different constraints. The symbols $h_a$, $l_a$, and $d_a$ are used to denote the head node, label, and dependent node, respectively, of an arc $a$ (that is, $a = (h_a, l_a, d_a)$); IN-SPAN($i$) is true if $i$ is contained in a span in $S_C$; END-SPAN($i$) is true if $i$ is the last word in a span in $S_C$; $s(i)$ denotes the span containing $i$ (with a dummy span for all words that are not contained in any span); $r(s)$ denotes the designated root of span $s$ (if any); #CC records the number of connected components in the current span up to and including the last word that was pushed onto the stack; NONE and ROOT are true if we allow no outgoing arcs from spans and if we allow outgoing arcs only from the span root, respectively.

some transition sequence (completeness).[2] In constrained parsing, we want to restrict the system so that, when applied to a sentence $x$, it is sound and complete for the subset of PDTs that satisfy all constraints.

## 3. Parsing with Arc Constraints

*Arc Constraints.* Given a sentence $x = w_1, \ldots, w_n$ and a label set $L$, an **arc constraint set** is a set $A_C$ of dependency arcs $(i,l,j)$ $(1 \leq i,j \leq n, i \neq j \neq n, l \in L)$, where each arc is required to be included in the parser output. Because the arc-eager system can only derive PDTs, the arc constraint set has to be such that the **constraint graph** $G_C = (V_x, A_C)$ can be extended to a PDT, which is equivalent to requiring that $G_C$ is a PDG. Thus, the task of arc-constrained parsing can be defined as the task of deriving a PDT $G$ such

2 Although the transition system in Nivre (2008) is complete but not sound, it is trivial to show that the system as presented here (with the root node at the end of the buffer) is both sound and complete.

that $G_C \subseteq G$. An arc-constrained transition system is sound if it only derives proper extensions of the constraint graph and complete if it derives all such extensions.

*Added Preconditions.* We know that the unconstrained arc-eager system can derive any PDT for the input sentence $x$, which means that any arc in $V_x \times L \times V_x$ is reachable from the initial configuration, including any arc in the arc constraint set $A_C$. Hence, in order to make the parser respect the arc constraints, we only need to add preconditions that block transitions that would make an arc in $A_C$ unreachable.[3] We achieve this through the following preconditions, defined formally in Figure 2 under the heading ARC CONSTRAINTS for each transition:

- LEFT-ARC$_l$ in a configuration $(\sigma|i,j|\beta,A)$ adds the arc $(j,l,i)$ and makes unreachable any arc that involves $i$ and a node in the buffer (other than $(j,l,i)$). Hence, we permit LEFT-ARC$_l$ only if no such arc is in $A_C$.
- RIGHT-ARC$_l$ in a configuration $(\sigma|i,j|\beta,A)$ adds the arc $(i,l,j)$ and makes unreachable any arc that involves $j$ and a node on the stack (other than $(i,l,j)$). Hence, we permit RIGHT-ARC$_l$ only if no such arc is in $A_C$.
- REDUCE in a configuration $(\sigma|i,j|\beta,A)$ pops $i$ from the stack and makes unreachable any arc that involves $i$ and a node in the buffer. Hence, we permit REDUCE only if no such arc is in $A_C$.
- SHIFT in a configuration $(\sigma,i|\beta,A)$ moves $i$ to the stack and makes unreachable any arc that involves $j$ and a node on the stack. Hence, we permit SHIFT$_l$ only if no such arc is in $A_C$.

*Complexity and Correctness.* Because the transitions remain the same, the arc-constrained parser will terminate after at most $2n$ transitions, just like the unconstrained system. However, in order to guarantee termination, we must also show that at least one transition is applicable in every non-terminal configuration. This is trivial in the unconstrained system, where the SHIFT transition can apply to any configuration that has a non-empty buffer. In the arc-constrained system, SHIFT will be blocked if there is an arc $a \in A_C$ involving the node $i$ to be shifted and some node on the stack, and we need to show that one of the three remaining transitions is then permissible. If $a$ involves $i$ and the node on top of the stack, then either LEFT-ARC$_l$ and RIGHT-ARC$_l$ is permissible (in fact, required). Otherwise, either LEFT-ARC$_l$ or REDUCE must be permissible, because their preconditions are implied by the fact that $A_C$ is a PDG.

In order to obtain linear parsing complexity, we must also be able to check all preconditions in constant time. This can be achieved by preprocessing the sentence $x$ and arc constraint set $A_C$ and recording for each node $i \in V_x$ its constrained head (if any), its leftmost constrained dependent (if any), and its rightmost constrained dependent (if any), so that we can evaluate the preconditions in each configuration without having to scan the stack and buffer linearly. Because there are at most $O(n)$ arcs in the arc constraint set, the preprocessing will not take more than $O(n)$ time but guarantees that all permissibility checks can be performed in $O(1)$ time.

Finally, we note that the arc-constrained system is sound and complete in the sense that it derives all and only PDTs compatible with a given arc constraint set $A_C$ for a sentence $x$. Soundness follows from the fact that, for every arc $(i,l,j) \in A_C$, the preconditions

3 For further discussion of reachability in the arc-eager system, see Goldberg and Nivre (2012, 2013).

force the system to reach a configuration of the form $(\sigma|\min(i,j),\max(i,j)|\beta,A)$ in which either LEFT-ARC$_l$ ($i > j$) or RIGHT-ARC$_l$ ($i < j$) will be the only permissible transition. Completeness follows from the observation that every PDT $G$ compatible with $A_C$ is also a PDG and can therefore be viewed as a larger constraint set for which every transition sequence (given soundness) derives $G$ exactly.

*Empirical Case Study: Imperatives.* Consider the problem of parsing commands to personal assistants such as Siri or Google Now. In this setting, the distribution of utterances is highly skewed towards imperatives making them easy to identify. Unfortunately, parsers trained on treebanks like the Penn Treebank (PTB) typically do a poor job of parsing such utterances (Hara et al. 2011). However, we know that if the first word of a command is a verb, it is likely the root of the sentence. If we take an arc-eager beam search parser (Zhang and Nivre 2011) trained on the PTB, it gets 82.14 labeled attachment score on a set of commands.[4] However, if we constrain the same parser so that the first word of the sentence must be the root, accuracy jumps dramatically to 85.56. This is independent of simply knowing that the first word of the sentence is a verb, as both parsers in this experiment had access to gold part-of-speech tags.

## 4. Parsing with Span Constraints

*Span Constraints.* Given a sentence $x = w_1, \ldots, w_n$, we take a **span constraint set** to be a set $S_C$ of non-overlapping spans $[i,j]$ ($1 \leq i < j \leq n$). The task of span-constrained parsing can then be defined as the task of deriving a PDT $G$ such that, for every span $[i,j] \in S_C$, $G_{[i,j]}$ is a (projective) spanning tree over the interval $[i,j]$. A span-constrained transition system is sound if it only derives dependency graphs compatible with the span constraint set and complete if it derives all such graphs. In addition, we may add the requirement that no word inside a span may have dependents outside the span (NONE), or that only the root of the span may have such dependents (ROOT).

*Added Preconditions.* Unlike the case of arc constraints, parsing with span constraints cannot be reduced to simply enforcing (and blocking) specific dependency arcs. In this sense, span constraints are more global than arc constraints as they require entire subgraphs of the dependency graph to have a certain property. Nevertheless, we can use the same basic technique as before and enforce span constraints by adding new preconditions to transitions, but these preconditions need to refer to variables that are updated dynamically during parsing. We need to keep track of two things:

- Which word is the designated root of a span? A word becomes the designated root $r(s)$ of its span $s$ if it acquires a head outside the span or if it acquires a dependent outside the span under the ROOT condition.
- How many connected components are in the subgraph over the current span up to and including the last word pushed onto the stack? A variable #CC is set to 1 when the first span word enters the stack, incremented by 1 for every SHIFT and decremented by 1 for every LEFT-ARC$_l$.

4 Data and splits from the Web Treebank of Petrov and McDonald (2012). Commands used for evaluation were sentences from the test set that had a sentence initial verb root.

Given this information, we need to add preconditions that guarantee the following:

- The designated root must not acquire a head *inside* the span.
- No word except the designated root may acquire a head *outside* the span.
- The designated root must not be popped from the stack before the last word of the span has been pushed onto the stack.
- The last word of a span must not be pushed onto the stack in a RIGHT-ARC$_l$ transition if #CC $> 1$.
- The last word of a span must not be pushed onto the stack in a SHIFT transition if #CC $> 0$.

In addition, we must block outside dependents of all words in a span under the NONE condition, and of all words in a span other than the designated root under the ROOT condition. All the necessary preconditions are given in Figure 2 under the heading SPAN CONSTRAINTS.

*Complexity and Correctness.* To show that the span-constrained parser always terminates after at most $2n$ transitions, it is again sufficient to show that there is at least one permissible transition for every non-terminal configuration. Here, SHIFT is blocked if the word $i$ to be shifted is the last word of a span and #CC $> 0$. But in this case, one of the other three transitions must be permissible. If #CC $= 1$, then RIGHT-ARC$_l$ is permissible; if #CC $> 1$ and the word on top of the stack does not have a head, then LEFT-ARC$_l$ is permissible; and if #CC $> 1$ and the word on top of the stack already has a head, then REDUCE is permissible (as #CC $> 1$ rules out the possibility that the word on top of the stack has its head outside the span). In order to obtain linear parsing complexity, all preconditions should be verifiable in constant time. This can be achieved during initial sentence construction by recording the span $s(i)$ for every word $i$ (with a dummy span for words that are not inside a span) and by updating $r(s)$ (for every span $s$) and #CC as described herein.

Finally, we note that the span-constrained system is sound and complete in the sense that it derives all and only PDTs compatible with a given span constraint set $S_C$ for a sentence $x$. Soundness follows from the observation that failure to have a connected subgraph $G_{[i,j]}$ for some span $[i, j] \in S_C$ can only arise from pushing $j$ onto the stack in a SHIFT with #CC $> 0$ or a RIGHT-ARC$_l$ with #CC $> 1$, which is explicitly ruled out by the added preconditions. Completeness can be established by showing that a transition sequence that derives a PDT $G$ compatible with $S_C$ in the unconstrained system cannot violate any of the added preconditions, which is straightforward but tedious.

*Empirical Case Study: Korean Parsing.* In Korean, white-space-separated tokens correspond to phrasal units (similar to Japanese *bunsetsus*) and not to basic syntactic categories like nouns, adjectives, or verbs. For this reason, a further segmentation step is needed in order to transform the space-delimited tokens to units that are a suitable input for a parser and that will appear as the leaves of a syntactic tree. Here, the white-space boundaries are good candidates for posing hard constraints on the allowed sentence structure, as only a single dependency link is allowed between different phrasal units, and all the other links are phrase-internal. An illustration of the process is given in Figure 3. Experiments on the Korean Treebank from McDonald et al. (2013) show that adding span constraints based on white space indeed improves parsing accuracy for an arc-eager beam search parser (Zhang and Nivre 2011). Unlabeled attachment score

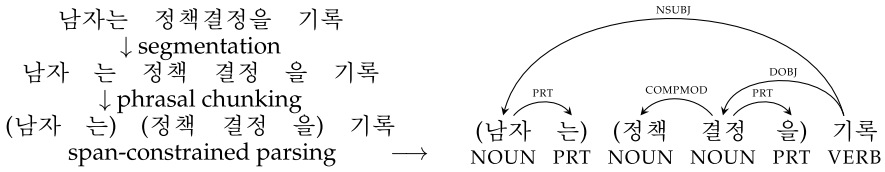


**Figure 3**
Parsing a Korean sentence (*the man writes the policy decisions*) using span constraints derived from original white space cues indicating phrasal chunks.

increases from an already high 94.10 without constraints to 94.92, and labeled attachment score increases from 89.91 to 90.75.

*Combining Constraints.* What happens if we want to add arc constraints on top of the span constraints? In principle, we can simply take the conjunction of the added preconditions from the arc constraint case and the span constraint case, but some care is required to enforce correctness. First of all, we have to check that the arc constraints are consistent with the span constraints and do not require, for example, that there are two words with outside heads inside the the same span. In addition, we need to update the variables $r(s)$ already in the preprocessing phase in case the arc constraints by themselves fix the designated root because they require a word inside the span to have an outside head or (under the ROOT condition) to have an outside dependent.

## 5. Conclusion

We have shown how the arc-eager transition system for dependency parsing can be modified to take into account both arc constraints and span constraints, without affecting the linear runtime and while preserving natural notions of soundness and completeness. Besides the practical applications discussed in the introduction and case studies, constraints can also be used as partial oracles for parser training.

## References

Ballesteros, Miguel and Joakim Nivre. 2013. Getting to the roots of dependency parsing. *Computational Linguistics*, 39:5–13.

Eisner, Jason and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 457–464, Santa Cruz, CA.

Eisner, Jason M. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 340–345, Copenhagen.

Goldberg, Yoav and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics*, pages 959–976, Shanghai.

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

Hara, Tadayoshi, Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2011. Exploring difficulties in parsing imperatives and questions. In *Proceedings of the 5th International Joint Conference on Natural Language Processing*, pages 749–757, Chiang Mai.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Koo, Terry and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1–11, Uppsala.

McDonald, Ryan, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 91–98, Ann Arbor, MI.

McDonald, Ryan, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. 2013. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia.

Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, pages 149–160, Nancy.

Nivre, Joakim. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553.

Nivre, Joakim, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning*, pages 49–56, Boston, MA.

Petrov, Slav and Ryan McDonald. 2012. Overview of the 2012 shared task on parsing the web. In *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)*, Montreal.

Yamada, Hiroyasu and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies*, pages 195–206, Nancy.

Zhang, Yue and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 562–571, Honolulu, HI.

Zhang, Yue and Joakim Nivre. 2011. Transition-based parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, pages 188–193, Portland, OR.