Proceedings of the 5th International Workshop on Parsing Technologies, pp. 54-65, Cambridge, MA, September 1997. [An improved and extended version appears in Harry C. Bunt and Anton Nijholt (eds.), *New Developments in Natural Language Parsing*, Kluwer, 2000.]

BILEXICAL GRAMMARS AND A CUBIC-TIME PROBABILISTIC PARSER*

Jason Eisner
Dept. of Computer and Information Science, Univ. of Pennsylvania
200 South 33rd St., Philadelphia, PA 19103-6389 USA
Email: jeisner@linc.cis.upenn.edu

1 Introduction

Computational linguistics has a long tradition of *lexicalized* grammars, in which each grammatical rule is specialized for some individual word. The earliest lexicalized rules were word-specific subcategorization frames. It is now common to find fully lexicalized versions of many grammatical formalisms, such as context-free and tree-adjoining grammars [Schabes *et al.* 1988]. Other formalisms, such as dependency grammar [Mel'čuk1988] and head-driven phrase-structure grammar [Pollard & Sag 1994], are explicitly lexical from the start.

Lexicalized grammars have two well-known advantages. Where syntactic acceptability is sensitive to the quirks of individual words, lexicalized rules are necessary for linguistic description. Lexicalized rules are also computationally cheap for parsing written text: a parser may ignore those rules that do not mention any input words.

More recently, a third advantage of lexicalized grammars has emerged. Even when syntactic *acceptability* is not sensitive to the particular words chosen, syntactic *distribution* may be [Resnik 1993]. Certain words may be able but highly unlikely to modify certain other words. Such facts can be captured by a *probabilistic* lexicalized grammar, where they may be used to resolve ambiguity in favor of the most probable analysis, and also to speed parsing by avoiding ("pruning") unlikely search paths. Accuracy and efficiency can therefore both benefit.

Recent work along these lines includes [Charniak 1995, Collins 1996, Eisner 1996b, Collins 1997], who reported state-of-the-art parsing accuracy. Related models are proposed without evaluation in [Lafferty *et al.* 1992, Alshawi 1996].

This recent flurry of probabilistic lexicalized parsers has focused on what one might call **bilexical grammars**, in which each grammatical rule is specialized for not one but *two* individual words.[1] The central insight is that specific words subcategorize to some degree for other specific words: *tax* is a good object for the verb *raise*. Accordingly, these models estimate, for example, the probability that (a phrase headed by) word y modifies word x, for any two words x, y in the vocabulary V.

At first blush, probabilistic bilexical grammars appear to carry a substantial computational penalty. Chart parsers derived directly from CKY or Earley's algorithm take time $O(n^3 \min(n, |V|)^2)$, which amounts to $O(n^5)$ in practice. Such algorithms implicitly or explicitly regard the grammar as a context-free grammar in which a noun phrase headed by *tiger* bears the special nonterminal $\mathrm{NP}_{\mathrm{tiger}}$. Such $\approx O(n^5)$ algorithms are explicitly used by [Alshawi 1996, Collins 1996], and almost certainly by [Charniak 1995] as well.

The present paper formalizes an inclusive notion of bilexical grammars, and shows that they can be parsed in time only $O(n^3 g^3 t^2 m) \approx O(n^3)$, where g, t, and m are bounded by the grammar and are typically small. (g is the maximum number of senses per input word, t measures the degree of lexical interdependence that the grammar allows *among* the several children of a word, and m bounds the number of modifier relations that the parser need distinguish for a given pair of words.) The new algorithm also reduces space requirements to $O(n^2 g^2 t) \approx O(n^2)$, from the $\approx O(n^3)$ required by CKY-style approaches to bilexical grammar. The parsing algorithm finds the highest-scoring analysis or analyses generated by the grammar, under a probabilistic or other measure. Non-probabilistic grammars may be treated as a special case.

* I am grateful to Mike Collins and Joshua Goodman for useful discussions of this work.

[1] Actually, [Lafferty *et al.* 1992] is formulated as a *trilexical* model, though the influence of the third word could be ignored.

The new $\approx O(n^3)$-time algorithm has been implemented, and was used in the experimental work of [Eisner 1996a, Eisner 1996b], which compared various bilexical probability models. The algorithm also applies to the head-automaton models of [Alshawi 1996] and the phrase-structure models of [Collins 1996, Collins 1997], allowing $\approx O(n^3)$-time rather than $\approx O(n^5)$-time parsing, granted the (linguistically sensible) restrictions that the number of distinct X-bar levels is bounded and that left and right adjuncts are independent of each other.

2 Formal Definition of Bilexical Grammars

2.1 Unweighted Bilexical Grammars

A bilexical grammar consists of the following elements:

- A set V of words, called the **vocabulary**, which contains a distinguished symbol ROOT.

  The elements of V may be used to represent word senses: so V can contain separate elements $bank_1, bank_2$, and $bank_3$ to represent the various meanings of *bank*. For parsing to be efficient, the maximum number of senses per word, g, should be small.

- A set M of one or more **modifier roles**.

  M does not affect the set of sentences generated by the grammar, but it affects the structures assigned to them. In these structures, elements of M will be used to annotate syntactic or semantic relations among words. For example, *John* might not merely modify *loves*, but modify it as SUBJECT or OBJECT, or as AGENT or PATIENT; these are roles in M.

  For parsing to be efficient, M should be small. (While a semantic theory may posit a great many modifier roles, it need not be the parser's job to make the finer distinctions.) More precisely, what should be small is m, the maximum number of modifier roles available for connecting two *given* words, such as *John* and *loves*.

- For each word w, a pair of deterministic finite-state automata ℓ_w and r_w. Each automaton accepts some set of strings over the alphabet $V \times M$.

  ℓ_w specifies the possible sequences of left dependents (arguments and adjuncts) for w. r_w specifies the possible sequences of right dependents for w. By convention, the first element in such a sequence is closest to w in the surface string. Thus, the possible dependent sequences are specified by $\mathcal{L}(\ell_w)^R$ and $\mathcal{L}(r_w)$ respectively.

  Note that the collection of automata in a grammar may be implemented as a pair of functions ℓ and r, such that $\ell(w, s, w', \mu)$ returns the destination state of the (w', μ)-labeled arc that leaves state s of automaton ℓ_w, and similarly for $r(w, s, w', \mu)$. These functions may be computed in any convenient way.

  For parsing to be efficient, the maximum number of states per automaton, t, should be small. However, without penalty there may be arbitrarily many distinct automata (one per word in V), and each automaton state may have arbitrarily many *arcs* (one per possible dependent in V), so $|V|$ does not affect performance.

We must now define the language generated by the grammar, and the structures that the grammar assigns to sentences of this language.

Let a **dependency tree** be a rooted tree whose nodes (internal and external) are labeled with words from V, and whose edges are labeled with modifier roles from M. The children ("dependents") of a node are ordered with respect to each other and the node itself, so that the node has both **left children** that precede it and **right children** that follow it. Figure 1a illustrates a dependency tree all of whose edges are labeled with the empty string.

Given a bilexical grammar, a leaf x of a dependency tree may be **expanded** to modify the tree as follows. Let w be the word that labels x, α be any string of left dependents accepted by ℓ_w, and β be any string of right dependents accepted by r_w. These strings are in $(V \times M)^*$. Add $|\alpha|$ left children and $|\beta|$ right children to the leaf x. Label the left children and their attached edges with the symbol pairs in α (from right to left); similarly label the right children and their attached edges with the symbol pairs in β (from left to right).

A **dependency parse tree** is a dependency tree generated from the grammar by starting with a single node, labeled with the special symbol ROOT $\in V$, and repeatedly expanding leaves until every node has been expanded exactly once. Figure 1a may be so obtained: one typical expansion step gives the leaf *plan* the child sequences $\alpha =$ *the*, $\beta =$ *of, raise*. Another such step expands *the*, choosing to give it no children at all ($\alpha = \beta = \epsilon$).

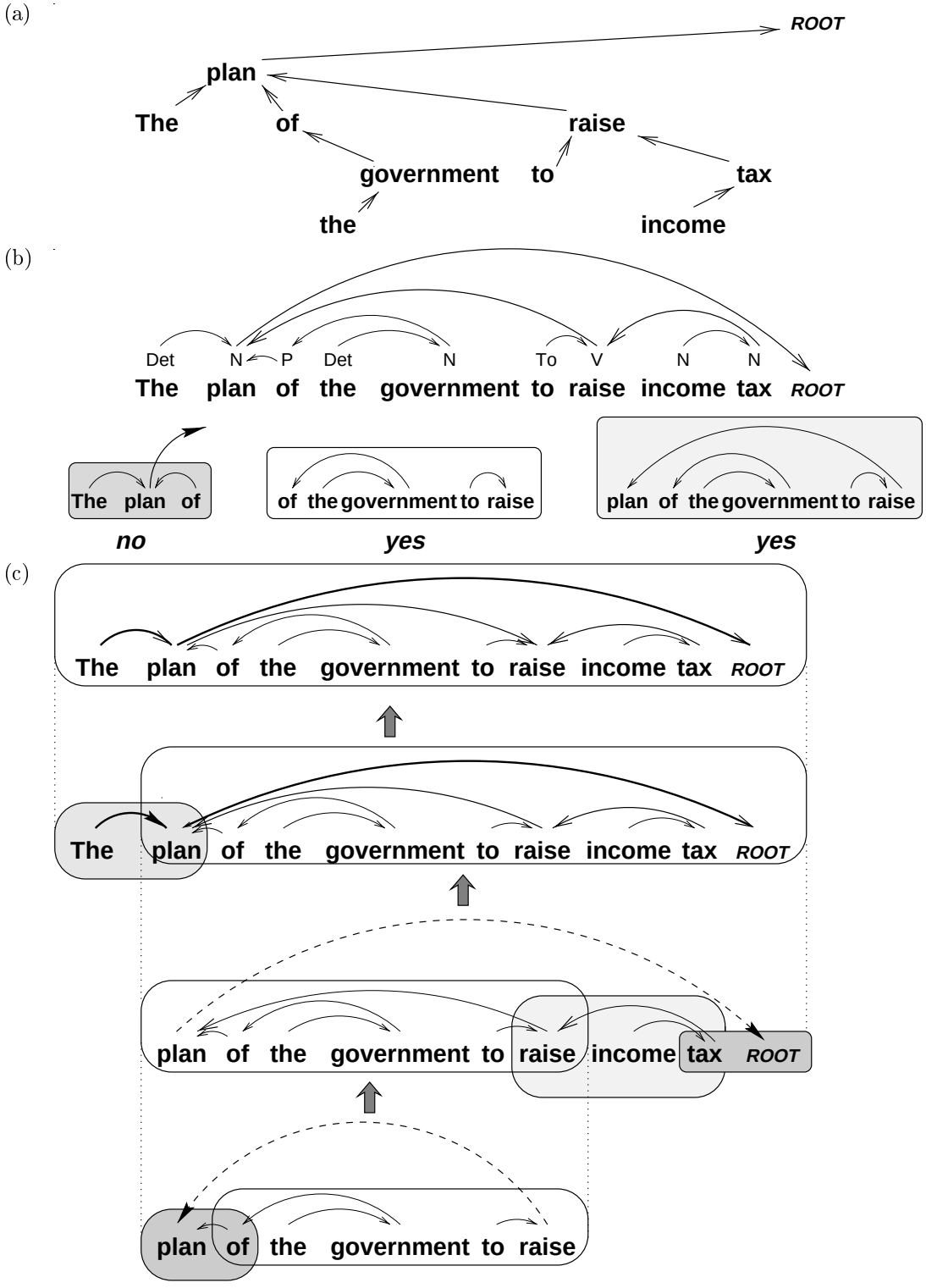


Figure 1: (a) An unlabeled dependency parse tree. (b) The same tree shown flattened out. A span of the tree is any substring such that no interior word of the span links to any word outside the span. One non-span and two spans are shown. (c) A span may be decomposed into smaller spans as repeatedly shown; therefore, a span can be built from smaller spans by following the arrows upward. The parsing algorithm builds successively larger spans in a dynamic programming table (chart). The minimal spans, used to seed the chart, are linked or unlinked word bigrams, such as *The*→*plan* or *tax* ROOT.

A string $\omega \in V^*$ is generated by the grammar, with analysis T, if T is a dependency parse tree and listing the node labels of T in infix order yields the string ω followed by ROOT. ω is called the **fringe** of T.

The term *bilexical* refers to the fact that (i) each $w \in V$ may specify a wholly different choice of automata ℓ_w and r_w, and furthermore (ii) each such automaton may make distinctions among individual words that are appropriate to serve as *children* of w. Thus the grammar is sensitive to specific *pairs* of lexical items. For example, it is possible for one lexical verb to select for a completely idiosyncratic set of nouns as subject, and another lexical verb to select for an entirely different set of nouns. Since it never requires more than a two-state automaton (though with many arcs!) to specify the set of possible subjects for a verb, there is no penalty for such behavior in the parsing algorithm to be described here.

2.2 Weighted Bilexical Grammars

The ability of a verb to subcategorize for an idiosyncratic set of nouns, as above, can be used to implement black-and-white ("hard") selectional restrictions. Where bilexical grammars are really useful, however, is in capturing *gradient* ("soft") selectional restrictions. A weighted bilexical grammar can equip each verb with an idiosyncratic *probability distribution* over possible object nouns, or indeed possible dependents of any sort. We now formalize this notion.

A **weighted finite-state automaton** A is a finite-state automaton that associates a real-valued **weight** with each arc and each final state. Following heavily-weighted arcs is intuitively "good," "probable," or "common"; so is stopping in a heavily-weighted final state. Each accepting path through A is automatically assigned a weight, namely, the sum of all arc weights on the path and the final-state weight of the last state on the path. Each string accepted by A is assigned the weight of its accepting path.

Now, we may define a weighted bilexical grammar as a bilexical grammar in which all the automata ℓ_w and r_w are weighted automata. The grammar assigns a weight to each dependency parse tree: namely, the sum of the weights of all strings of dependents, α and β, generated while expanding nodes to derive the tree. (This weight is well-defined: it does not depend on the order in which leaves were expanded.)

The goal of the parser is to determine whether a string $\omega \in V^*$ can be generated by the grammar, and if so, to determine the highest-weighted analysis for that sentence. It is convenient to set up the weighted automata so that each automaton formally accepts *all* strings in V^*, but assigns a weight of $-\infty$ to any that are not permitted by the competence grammar. Then a sentence is rejected as ungrammatical if its highest-weight analysis has weight only $-\infty$. The unweighted case is therefore a special case of the weighted case.

2.3 Lexical Selection

The above formalism must be extended to deal with lexical selection. Regrettably, the input to a parser is typically not a string in V^*. Rather, it contains ambiguous tokens such as *bank*, whereas the "words" in V are word senses such as $bank_1, bank_2$, and $bank_3$, or part-of-speech-tagged words such as *bank*/N and *bank*/V. One would like a parser to resolve these ambiguities as well as structural ambiguities.

We may modify the formalism as follows. Consider the unweighted case first. Let Ω be the real input—a string not in V^* but rather in $\mathcal{P}(V)^*$, where $\mathcal{P}$ denotes powerset. Thus the ith symbol of Ω is a **confusion set** of possibilities for the ith word of the input, e.g., $\{bank_1, bank_2, bank_3\}$. Ω is generated by the grammar, with analysis T, if some string $\omega \in V^*$ is so generated, where ω is formed by replacing each set in Ω with one of its elements. Note that ω is the fringe of T.

For the weighted case, each confusion set in the input string Ω assigns a weight to each of its members. Again, intuitively, the heavily-weighted members are the ones that are commonly correct, so the noun *bank*/N would be weighted more highly than the verb *bank*/V. We score dependency parse trees as before, except that now we also add to a tree's score the weights of all its fringe words, as selected from their respective confusion sets. Formally, we say that $\Omega = W_1 W_2 \ldots W_n \in \mathcal{P}(V)^*$ is generated by the grammar, with analysis T and weight $p + q_1 + \cdots + q_n$, if some string $\omega = w_1 w_2 \ldots w_n \in V^*$ is generated with analysis T and weight p, and $|\omega| = |\Omega| = n$ and for each $1 \leq i \leq n$, ω_i appears in the set W_i with weight q_i.

2.4 Adding String-Local Constraints

An extension is to allow the grammar to specify a list of **excluded bigrams**. If a tree's fringe contains an excluded bigram (two-word sequence), the tree is not considered a valid dependency parse tree, and is given

score $-\infty$.

§3.8 shows how this extension lets the scoring model consider such factors as the probability of the tag k-grams that appear in the parse (even for $k > 2$), as proposed by [Lafferty *et al.* 1992]. Consideration of such factors has been shown useful for probabilistic systems that simultaneously optimize tagging and parsing [Eisner 1996b].

3 Uses of Bilexical Grammars

Bilexical grammars, and the new parsing algorithm for them, are not limited to dependency-style structures. They are flexible enough to capture a variety of grammar formalisms and probability models. This section will illustrate the point with some key cases. We begin with the dependency case, and progress to phrase-structure grammars.

3.1 A Simple Case: Monolexical Dependency Grammar

It is straightforward to encode dependency grammars such as those of [Gaifman 1965]. We focus here on the case that [Milward 1994] calls Lexicalized Dependency Grammar (LDG). Milward demonstrates a parser for this case that requires $O(n^3g^3t^3) \approx O(n^3)$ time and $O(n^2g^2t^2) \approx O(n^2)$ space, using a left-to-right algorithm that maintains its state as an acyclic directed graph. Here t is taken to be the maximum number of dependents on a word. m does not appear because Milward takes tree edges to be unlabeled, i.e., $m = 1$.

LDG is defined to be only *monolexical*. Each word sense entry in the lexicon is for a word tagged with the type of phrase it projects. An entry for *helped*/S, which appears as head of the sentence *Nurses helped John wash*, may specify that it wants a left dependent sequence of the form w_1/N and a right dependent sequence of the form w_2/N, w_3/V. However, under LDG it cannot constrain the lexical content of w_1, w_2, or w_3, either discretely or probabilistically.[2]

By encoding a monolexical LDG as a bilexical grammar, and applying the algorithm described below in §4, we can improve parsing time and space by a factor of t. The encoding is straightforward. To capture the preferences for *helped*/S as above, we define $\ell_{helped/S}$ to be a two-state automaton that accepts exactly the set of nouns, and $r_{helped/S}$ to be a three-state automaton that accepts exactly those word sequences of the form (noun, verb). Obviously, $\ell_{helped/S}$ includes a great many arcs—one arc for every noun in V. This does not however affect parsing performance, which depends only on the number of *states* in the automaton.

3.2 Optional and Iterated Dependents

The use of automata makes the bilexical grammar considerably more flexible than its LDG equivalent. In the example of §3.1, $r_{helped/S}$ can be trivially modified so that the dependent verb is optional (*Nurses helped John*). LDG can accomplish this only by adding a new lexical sense of *helped/S*, increasing g.

Similarly, under a bilexical grammar, $\ell_{nurses/N}$ can be specified to accept dependent sequences of the form (adj, adj, adj, ... adj, (det)). Then *nurses* may be expanded into *weary Belgian nurses*. Unbounded iteration of this sort is not possible in LDG, where each word sense has a fixed number of dependents. In LDG, as in categorial grammars, *weary Belgian nurses* would have to be headed by the adjunct *weary*. Thus, even if LDG were sensitive to bilexicalized dependencies, it would not recognize *nurses*→*helped* as such a dependency.

3.3 Bilexical Dependency Grammar

In the example of §3.1, we may arbitrarily weight the individual noun arcs of the ℓ_{helped} automaton, according to how appropriate those nouns are as subjects of *helped*. (In the unweighted case, we might choose to rule out inanimate subjects altogether, by removing their arcs or assigning them the weight $-\infty$.) This turns the grammar from monolexical to bilexical, without affecting the cubic-time cost of the parsing algorithm of §4.

[2] What would happen if we tried to represent bilexical dependencies in such a grammar? In order to restrict w_2 to nouns denoting helpless animate creatures, the grammar would need a new nonterminal symbol, $\mathrm{NP}_{\mathrm{helpable}}$. All nouns in this class would then need additional lexical entries to indicate that they are possible heads of $\mathrm{NP}_{\mathrm{helpable}}$. The proliferation of such entries would drive g up to $|V|$ in Milward's algorithm, resulting in performance of $O(n^3|V|^3t^3)$ (or by ignoring rules that do not refer to lexical items in the input sentence, $O(n^6t^3) \approx O(n^6)$).

3.4 Probabilistic Bilexical Dependency Grammars

[Eisner 1996b] compares several probability models for dependency grammar. Each model simultaneously evaluates the part-of-speech tags and the dependencies in a given dependency parse tree. Given an untagged input sentence, the goal is to find the dependency parse tree with highest probability under the model.

Each of these models can be accomodated to the bilexical parsing framework, allowing a cubic-time solution. In each case, V is a set of part-of-speech-tagged words. For simplicity, M is taken to be a singleton set $\{\epsilon\}$. Each weighted automaton ℓ_w or r_w is defined so that it accepts any dependent sequence in V^*, but the automaton has 8 states, which enable interactions among successive dependents in a sequence. Any arc that accepts a noun (say) terminates in the Noun state. The w arc from Noun may be weighted differently than the w arcs from other states; so a given dependent word w may be more or less likely depending on whether the previous dependent in the sequence was a noun. The final-state weight of Noun may also be selected freely: so the sequence of dependents might be likely or unlikely to end with a noun.[3]

As sketched in [Eisner 1996a], each of Eisner's probability models is implemented as a particular scheme for weighting such an automaton. For example, model C regards ℓ_w and r_w as Markov processes, where each state specifies a probability distribution over its exit options, namely, its outgoing arcs and the option of halting. The weight of an arc or a final state is then the log of its probability. Thus if $r_{helped/V}$ includes an arc labeled with $(bathe/\mathrm{V}, \epsilon)$ and this arc is leaving the Noun state, then the arc weight is (an estimate of)

$$\log \Pr(\text{next right dependent is } \textit{bathe}/\mathrm{V} \mid \text{parent is } \textit{helped}/\mathrm{V} \text{ and previous right dependent was a noun})$$

The weight of a dependency parse tree under this probability model is a product of such factors, which means that it estimates Pr(dependency links & input words) according to a generative model. By contrast, model D estimates Pr(dependency links | input words), using arc weights that are roughly of the form

$$\log \Pr(\textit{bathe}/\mathrm{V} \text{ is a right dep. of } \textit{helped}/\mathrm{V} \mid \text{both words appear in sentence and prev. right dep. was a noun})$$

which is similar to the probability model of [Collins 1996]. Some of the models evaluated also rely on weighted string-local constraints, as implemented in §3.8 below.

3.5 Probabilistic Bilexical Phrase-Structure Grammar

In some situations, one wishes a parser to evaluate phrase-structure trees rather than dependency parse trees. [Collins 1997] observes that since VP and S are both verb-headed, the dependency grammars of §3.4 would falsely expect them to appear in the same environments. (The expectation is false because *continue* subcategorizes for VP only.) Phrase-structure trees address the problem by providing nonterminal labels. In addition, phrase-structure trees are less flat than dependency trees: a word's dependents attach to it at different levels, providing an obliqueness order on the dependents of a word. Obliqueness is of semantic interest, and is also exploited by [Wu 1995], whose statistical translation model preserves the topology (ID, not LP) of binary-branching parses.

Fortunately, it is possible to encode (headed) phrase structure trees as dependency trees with labeled edges. One can therefore use the fast bilexical parsing algorithm of §4 to generate the highest-weighted dependency tree, and then convert that tree to a phrase-structure tree.

For the encoding, we expand V so that all words are tagged not with single nonterminals but with **nonterminal chains**. Let us encode the phrase-structure tree *[John [continued$_V$ [to bathe$_V$ himself]$_{VP}$]$_{VP}$]$_S$*. The word *bathe* is the head of V and VP constituents, while *continued* is the head of V, VP, and S constituents (reading from the leaves upward). In the dependency tree, we therefore tag them as *bathe/V,VP* and *continued/V,VP,S*. The automaton $r_{continued}$ can now require a dependent's tag to end with VP; this captures the ungrammaticality of **John continued [Oscar to bathe himself]*.

Next, we put M to be the set of nonterminals. To encode the fact that in the phrase-structure tree, *himself* modifies *bathe* by attaching at the VP level above *bathe*, we attach *himself* to *bathe* in the corresponding dependency tree with a VP-labeled edge.

Finally, we wish to ensure that any dependency tree the parser returns can be mapped back to a phrase-structure tree. For this reason, the bilexical grammar's automata must require that the left or right dependents of a word are appropriate to the word's tag. Thus, any edges depending from *continued/V,VP,S* must be

[3] The eight states are START, Noun, Verb, Noun Modifier, Adverb, Preposition, Wh-word, and Punctuation.

labeled with V, VP, or S—the nonterminals that *continued* projects—and must fall in this order on each side. For instance, $r_{continued/V,VP,S}$ may not allow S-labeled right dependents to precede VP-labeled right dependents.

For this scheme to work, certain conditions must hold of the phrase-structure trees we are encoding. Only finitely many nonterminal chains may be available to tag a given word, and the nonterminals in a chain may not repeat. This is essentially in conformance with X-bar theory. The one difference is in the treatment of adjunction: in X-bar theory, three adjuncts to a VP would require 4 VP levels. One may work around this by stipulating a single VP-ADJUNCT level above VP, to which all VP adjuncts (0 or more) attach.[4]

This encoding scheme improves on that of [Collins 1996], because any dependency tree can be converted back to a phrase-structure tree, and this tree is unique. This makes it possible to use the fast bilexical parser, which (unlike Collins's) produces dependency trees without regard to whether they were derived from phrase structure trees. It also means that a probabilistic parsing model (unlike Collins's) need not be deficient, i.e., probability is not assigned to dependency structures that cannot be used by the phrase-structure grammar.

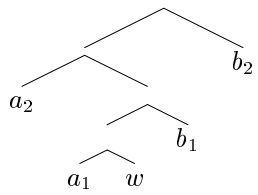
One interesting artifact of Collins's encoding scheme is that unary rules are impossible: every nonterminal level must have at least one dependent, whether on the left or on the right. If desired, the new scheme can be made to have this property as well. We augment V further, so that each nonterminal in a nonterminal chain is annotated with either "L" or "R," indicating that it requires children on the specified side. If VP is marked with "L," then the left-child automaton for that word must rule out any left-dependent sequence that does not have at least one dependent with a VP-labeled edge, and preserve the weights on all other dependent sequences.

3.6 Relationship to Head Automata

It should be noted that weighted bilexical grammars are essentially a special case of head-automaton grammars [Alshawi 1996]. As noted in the introduction, head-automaton grammars are bilexical in spirit. However, the left and right dependents of a word w are generated not separately, by automata ℓ_w and r_w, but in interleaved fashion by a single weighted automaton, d_w. d_w assigns weight to strings over the alphabet $V \times \{\leftarrow, \rightarrow\}$; each such string is an interleaving of lists of left and right dependents from V.

Head automata, as well as [Collins 1997], can model the case that §3.5 cannot: where nonterminal sequences may include nonterminal cycles. [Alshawi 1996] points out that this makes head automata are fairly powerful. An automaton corresponding to the regular expression $((a, \leftarrow)(b, \rightarrow))^*$ requires its word to have an equal number of left and right children, i.e,. $a^n w b^n$. (By contrast, a bilexical grammar or dependency grammar may be made to generate $\{a^n w b^n : n \geq 0\}$ only by making words other than w the heads of these strings, so that the words that are allowed to interact bilexically would change.)

For syntactic description, this added power is probably unnecessary. (Linguistically plausible interactions among left and right subcat frames, such as fronting, can be captured in bilexical grammars simply via multiple word senses.) What head-automaton grammars offer over bilexical grammars is the ability for a head to specify an obliqueness order over all its dependents, including arbitrarily many adjuncts. A head-automaton parse tree for $a_2 a_1 w b_1 b_2$ is more finely detailed than in dependency grammar. It essentially gives the phrase headed by w a binary-branching analysis, such as



.

3.7 Idiom Encoding

To the bilexical construction of §3.3, one may add detectors for special phrases. Consider the idioms (a) *run scared*, (b) *run circles [around NP]*, and (c) *run NP [into the ground]*. (a), like most idioms, is only bilexical, so it may be captured "for free": simply increase the weight of the *scared* arc in $r_{run/V}$. But because (b) and (c) are trilexical, they require augmentation to the grammar. (b) requires a special state to be added to $r_{run/V}$, so that the dependent sequence (*circles*, *around*) may be recognized and weighted heavily. (c) requires a specialized lexical entry for *into*; this sense is a preferred dependent of *run* and has *ground* as a preferred dependent.

[4] This scheme conflates the two semantically distinct parses of "intentionally knock twice"; a later semantic component would have to resolve the scope ambiguity. The more powerful head automata of §3.6 could distinguish the readings, but at the cost of using a $O(n^5)$ rather than $O(n^3)$ parsing algorithm.

3.8 k-gram Tagging

One may use the string-local constraints of §2.4 to score dependency parse trees according to the trigram part-of-speech tagging model of [Church 1988]. Each input word w (including ROOT) is regarded as a confusion set over all tuples of the form (t'', t', t, w), where t is a tag for w and t'', t' are tags for the two words that precede w. (Thus, V consists of such tuples.) The weight of the tuple (t'', t', t, w) in its confusion set is $\log(\Pr(w \mid t) \cdot \Pr(t \mid t'', t'))$. The bigram $(t''_i, t'_i, t_i, w_i)(t''_{i+1}, t'_{i+1}, t_{i+1}, w_{i+1})$ is an excluded bigram unless $t'_i = t''_{i+1}$ and $t_i = t'_{i+1}$.

Because of the excluded bigrams, any dependency parse tree has a fringe (including ROOT) of the form

$$(\text{BOS}, \text{BOS}, t_1, w_1)(\text{BOS}, t_1, t_2, w_2)(t_1, t_2, t_3, w_3) \ldots (t_{n-1}, t_n, \text{EOS}, \text{ROOT})$$

Then the total weight accruing to the tree from the confusion sets is

$$\log(\Pr(t_1 \mid \text{BOS}, \text{BOS}) \cdot \Pr(t_2 \mid \text{BOS}, t_1) \cdot \Pr(t_3 \mid t_1, t_2) \cdots \Pr(\text{EOS} \mid t_{n-1}, t_n)$$

$$\cdot \Pr(w_1 \mid t_1) \cdots \Pr(w_n \mid t_n) \cdot \Pr(\text{ROOT} \mid \text{EOS}))$$

and to maximize this total is to maximize the probability product shown, just as [Church 1988] does. k-grams for $k > 3$ may be handled in exactly the same way.

Since the parser maximizes the above sum of confusion-set weights *and* the weights accruing from the choice of paths through the automata, grammatical structure also helps determine the highest-weighted parse.

4 Cubic-Time Parsing

This section begins by reviewing the general idea of chart parsing, presenting a general method drawn from context-free "dotted-rule" methods such as [Graham *et al.* 1980, Earley 1970]. Second, we will see why this method is inefficient when applied to bilexical grammars in the obvious way. Finally, a more efficient (cubic-time) algorithm is presented, which applies the general chart parsing method rather differently.

4.1 Generalized Chart Parsing Method

The input is a string $\Omega = W_1 W_2 \ldots W_n$ of confusion sets (so each $W_i \subseteq V$). C (the **chart**) is an $(n+1) \times (n+1)$ array. The chart **cell** $C_{i,j}$ holds a set of **partial analyses** of the input. Each partial analysis has a weight, and also a **signature** that concisely describes its ability to combine with other partial analyses. For each signature s, the **subcell** $C_{i,j}[s]$ holds the highest-weight partial analysis of the input substring $W_{i+1} W_{i+2} \ldots W_j$ for which s is the signature.[5]

Let Discover(i, j, P) be an operation that replaces $C_{i,j}[signature(P)]$ with the partial analysis P if P has higher weight than the partial analysis currently in $C_{i,j}[signature(P)]$.

A basic chart-parsing method is then as follows:

```
1.  for i := 1 to n + 1
2.      foreach w ∈ W_i, where W_{n+1} = {ROOT}        (* seed chart with members of the confusion set *)
3.          foreach partial analysis a of the single word w
4.              Discover(i − 1, i, a)
5.  for width := 1 to n + 1
6.      for start := 0 to (n + 1) − width
7.          end := start + width
```

[5] In a more general conception, $C_{i,j}$[s] holds a summary of *all* known partial analyses of $W_{i+1} W_{i+2} \ldots W_j$ having signature s. Summaries must be defined in such a way that if f is an operation that combines two partial analyses, and A and B are sets of partial analyses of adjacent substrings, then $summary(\{f(a, b) : a \in A \text{ and } b \in B\})$ must be computable from $summary(A)$ and $summary(B)$. In addition, if A and B are both sets of partial analyses of $W_{i+1} W_{i+2} \ldots W_j$ having signature s, then $summary(A \cup B)$ must be computable from $summary(A)$ and $summary(B)$, so that new analyses can be added to the chart.

In the usual case, where the ultimate goal of the parser is to find the highest-weight dependency parse tree, $summary(A)$ is just the highest-weight parse tree in A (or, more generally, a forest of all parse trees in A that *tie* for the highest weight). However, if the parser is to return something else, one might set things up so that $summary(A)$ was a list of the 10 highest-weight parse trees in A—or even something more outré. If the parser is being used only to do language modeling for speech recognition, for instance, and the goal is to minimize the per-word error rate, then the summary might give a posterior probability distribution over the confusion set for the kth word, for each $i < k \leq j$; this would be determined by allowing the partial analyses in A to vote in proportion to their weight.

```
8.          for mid := start + 1 to end − 1
9.              foreach partial analysis a in C_{start,mid}        (* i.e., each C_{start,mid}[s] that is defined *)
10.                 foreach partial analysis b in C_{mid,end}
11.                     foreach way of combining a and b into a new weighted analysis c (if any)
12.                         Discover(start, end, c)
13.     foreach partial analysis a in C_{0,n+1}
14.         if signature(a) indicates that a is a full (not partial) analysis of the sentence
15.             then print a
```

The iterations in lines 3 and 11 have yet to be defined, but the dynamic programming idea is clear (and familiar): analyses of length-1 substrings can be created from the substrings themselves (line 3), and analyses of successively longer substrings can be created by gluing together shorter analyses in pairs (line 11), until we have one or more analyses of the whole sentence.

In particular, the problem has the optimal substructure property: any *optimal* analysis of a long string can be found by gluing together just *optimal* analyses of shorter substrings. (An optimal analysis is defined to be a partial analysis of maximum weight for its signature; Discover() ensures that the chart contains only optimal analyses.) For suppose that a and a' are partial analyses of the same substring, and have the same signature, but a has less weight than a'. Then suboptimal a cannot be part of any optimal analysis b in the chart—for if it were, the definition of signature ensures that we could substitute a' for a in b to get an partial analysis b' of greater total weight than b and the same signature as b, which contradicts b's optimality.

The algorithm's running time is dominated by the six nested loops, yielding time $\Theta(n^3S^2d)$. Here S is the maximum number of possible signatures that may fall in a given chart cell, and d is the maximum number of ways to combine two adjacent partial analyses into a larger one.

4.2 Inefficient Chart Parsing of Bilexical Grammars

How might we apply the above method to parsing of bilexical grammars? The obvious way is for each partial analysis to represent a subtree. More precisely, each partial analysis would represent a kind of dotted subtree that may not yet have acquired all its children. The signature of such a dotted subtree is a triple (w, s_{ℓ_w}, s_{r_w}), where $w \in V$ is an input word, s_{ℓ_w} is a state of ℓ_w, and s_{r_w} is a state of r_w. If both s_{ℓ_w} and s_{r_w} are final states, then the signature is said to be *complete*.

It is clear that in line 3 of the algorithm, the sole analysis a is the triple (w, start state of ℓ_w, start state of r_w). It is also clear that line 14 should ensure that we print only trees with complete signatures as analyses of the sentence. Finally, consider line 11. If a has signature (w, s_{ℓ_w}, s_{r_w}) and b has a complete signature, then there are m possible values of c in which b is attached to the root of the dotted subtree a as a new right child. These analyses have signatures

$$\{(w, s_{\ell_w}, r(w, s_{r_w}, w', \mu)) : \mu \in M\} \qquad \text{(where } r(w, \ldots) \text{ is the transition function of } r_w \text{ as defined in §2.1)}$$

The case where a is attached to the root of b as a new left child is similar, and may give another m values for c.

Why is this method inefficient? Because there are too many possible signatures. The probability with which b attaches to a depends on the roots of both a and b. Since the root w of a could be any of the words at positions $start + 1, start + 2, \ldots mid$, and there may be $\min(n, |V|)$ distinct such words in the worst case, the number S of possible signatures for a is at least $\min(n, |V|)$. The same is true for b, whose root w' could likewise be any of many words. But then the runtime of the algorithm is $\Omega(n^3 \min(n, |V|)^2 |M|) \approx \Omega(n^5)$. In a nutshell, the problem is that each chart cell may have to maintain many differently-headed analyses.

4.3 Efficient Chart Parsing of Bilexical Grammars

To eliminate these two additional $\min(n, |V|)$ factors, we must reduce the number of possible signatures for a partial analysis. The solution is for partial analyses to represent some kind of contiguous string other than constituents. Each partial analysis in $C_{i,j}$ will be a new kind of object called a span, which consists of one or two "half-constituents" in a sense to be described. The headword(s) of a span in $C_{i,j}$ are *guaranteed* to be at positions i and j in the sentence. This guarantee means that where $C_{i,j}$ in the previous section had n-fold uncertainty about the correct location of the headword for the optimal analysis of $W_{i+1}W_{i+2}\ldots W_j$, here it will have only 3-fold uncertainty. The three possibilities are that w_i is an unattached headword, that w_j is, or that both are.

Given a dependency parse tree, we know what its constituents are: a constituent is any substring consisting of a word and all its descendants. The inefficient parsing algorithm of the §4.2 assembled the correct parse tree by finding and gluing together analyses of the tree's constituents in an approved way. For something similar to be possible with spans, we must define what the spans of a given dependency parse tree are, and how to glue analyses of spans together into analyses of larger spans. Not every substring of the sentence is a correct constituent, and in the same way, not every substring is a correct span.

Figure 1a–b illustrates what spans are. A span of the dependency parse tree in (a) and (b) is any substring $w_i w_{i+1} \ldots w_j$ $(j > i)$ of the tree's fringe, such that none of the interior words of the span communicate with any words outside the span. Formally: if $i < k < j$, and w_k is a dependent of $w_{k'}$ or vice-versa, then $i \leq k' \leq j$.

Since we will build the parse by assembling analyses of spans, and the interiors of adjacent spans are insulated from each other, we crucially never need to know anything about the internal analysis inside a span. When we combine two adjacent spans, we never add a link from or to the interior of either. For, by the definition of span, if such a link were necessary, then the spans being combined could not be spans of the true parse anyway. There is always some other way of decomposing the true parse (itself a span) into smaller spans so that no such links from or to interiors are necessary.

Figure 1c shows such a decomposition.[6] Any span of greater than two words, say again from w_i to w_j, can be decomposed uniquely by the following deterministic procedure. Choose $i < k < j$ such that w_k is the rightmost word (strictly inside the span) that connects to w_i; if there is no such word, put $k = i+1$. Because crossing links are not allowed, the substrings from $w_i \ldots w_k$ and $w_k \ldots w_j$ must also be spans. We can therefore assemble the original $w_i \ldots w_j$ span by concatenating the $w_i \ldots w_k$ and $w_k \ldots w_j$ spans, and optionally adding a link between the end words, w_i and w_j. By construction, there is never any need to add a link between any other pair of words. Notice that when the two narrower spans are concatenated, w_k gets its left children from one span and its right children from the other.

The procedure for choosing k can be rephrased declaratively. To wit, the left span in the concatenation, $w_i \ldots w_k$, must be **simple** in the following sense: it must have a direct link between w_i and w_k, or else have only two words.

It is useful to note that the analysis of a span always takes one of three forms; Figure 1b illustrates the first two (labeled "yes"). In the first case, the endwords w_i and w_j are not yet connected to each other: that is, the path between them in the final parse tree will involve words outside the span. Then the span consists of two "half-constituents"—w_i with all its right descendants, followed by w_j with all its left descendants. w_i and w_j both need parents. In the second case, w_j is a descendant of w_i via a chain of one or more leftward links within the span itself; then the span consists of w_i and all its right descendants to the left of w_j (inclusive), and only w_i still needs a parent. The third case is the mirror image of the second. (It is impossible for both w_i and w_j to both have parents inside the span: for then some word interior to the span would need a parent outside it.)

The signature of a span does not have to state anything about the internal analysis except which of these three cases holds—i.e., which of w_i, w_j need parents. This is needed so that the parser knows when it is able to add a link from i or j to a more distant word after concatenation, without creating multiple parents or otherwise jeopardizing the form of the dependency parse. To determine the allowability of the dependent introduced by such a new link, or the weight associated with it, the signature of a span from w_i to w_j must also include the states of the automata r_{w_i} and ℓ_{w_j}.

We can now understand the actual algorithm. It is convenient to slightly alter the definition of $C_{i,j}$, so that it stores the best analysis (as a span) of $W_i W_{i+1} \ldots W_j$.[7] We may represent an analysis of a span as a tuple $(linktype, a, b)$, where $linktype$ specifies the label $(\in M)$ and direction of the link, if any, between the leftmost and rightmost words of the span, and where a and b point to the narrower spans that were concatenated to obtain this one. If the span is only two words wide, then we represent an analysis of it as $(linktype, w_1, w_2)$ so that the analysis specifies the words it has chosen from the confusion set.

As usual, the analyses we discover in a cell of the chart are organized into competitions or subcells by their signatures. The signature of an analysis has the form $(L?, R?, w_L, w_R, s_r, s_\ell, simple?)$. Here $L?$ and $R?$ are boolean variables stating whether the leftmost and rightmost words have parents in the span. w_L and w_R are the leftmost and rightmost words of the span (as chosen from the appropriate confusion sets). s_r is the state

[6] [Lafferty *et al.* 1992] give a related decomposition for the case of link grammar, and use it to construct an $O(n^3)$ top-down parsing algorithm. The bilexical parsing algorithm could be adapted to the case of link grammars, in which case it would resemble a bottom-up version of the independent algorithm of [Lafferty *et al.* 1992].

[7] Why not start with W_{i+1} as in §4.1? Because the spans we concatenate, unlike constituents that we concatenate, overlap in one word.

of the leftmost word's right-dependent automaton r_{w_L} after the automaton has read all the leftmost word's dependents within the span, and s_ℓ is similarly the state of the rightmost word's left-dependent automaton ℓ_{w_R}. Finally, *simple*? is true just if the span is simple, as defined above; we use this to prevent ourselves from finding the same analysis in multiple ways.

```
1.  for i := 1 to n
2.      foreach w_i ∈ W_i, w_{i+1} ∈ W_{i+1} such that w_i w_{i+1} is not an excluded bigram, where W_{n+1} = {ROOT}
3.          foreach linktype ∈ ({←, →} × M) ∪ {NONE}
4.              Discover(i, i+1, (linktype, w_i, w_{i+1}))        (* seed with two-word spans *)
5.  for width := 3 to n+1
6.      for start := 1 to (n+1) − width
7.          end := start + width
8.          for mid := start + 1 to end − 1
9.              foreach partial analysis a in C_{start,mid}        (* i.e., a is each C_{start,mid}[s] that is defined *)
10.                 if sig(a).simple?        (* where sig(a) is just s from comment above; don't recompute it *)
11.                     foreach partial analysis b in C_{mid,end} such that sig(b).w_L = sig(a).w_R
                                                                   (* so a, b agree on sense of overlapping word *)
12.                         if (sig(a).R? xor sig(b).L?)        (* overlapping word gets exactly one parent *)
13.                             Discover(start, end, (NONE, a, b))        (* version without a covering link *)
14.                             if not (sig(a).L? or sig(b).R?)        (* prevents multiple parents, link cycles *)
15.                                 foreach linktype ∈ {←, →} × M
16.                                     Discover(start, end, (linktype, a, b))
17. foreach partial analysis a in C_{0,n+1}
18.     if sig(a).L?        (* so a is a connected tree rooted at ROOT *)
19.         then print a
```

Computing the signatures is fairly straightforward. For example, if *linktype* is rightward in line 16 (making w_{start} a dependent of w_{end}), then the new analysis $(linktype, a, b)$ has the signature

$$(\text{TRUE}, \text{FALSE}, sig(a).w_L, sig(b).w_R, sig(a).s_r, \delta(sig(b).s_\ell, sig(a).w_L), \text{TRUE})$$

Computing weights of analyses is also fairly straightforward. In the above example, $(linktype, a, b)$ has weight

$$weight(a) + weight(b) + arcweight(sig(b).s_\ell, sig(a).w_L) + stopweight(sig(a).s_\ell) + stopweight(sig(b).s_r)$$

Note the use of the final-state weights, *stopweight*, to reflect the fact that the overlapping word w_{mid} (see line 11) can no longer get new children once it is hidden in the interior of the span.[8] For a two-word span $(linktype, w_i, w_{i+1})$ with rightward link, as created in line 4, the weight is $arcweight(w_{i+1}.\text{START}, w_i)$ plus the weight of w_i (only!) in its confusion set.

The algorithm requires $O(n^2S)$ space for the chart, where S is the maximum number of signatures per cell. The running time is again dominated by the six nested loops. It is $O(n^3 \cdot S^2 \cdot |M|)$. Given the definition of signatures, $S = O(g^2t^2)$; that is, it is bounded by a constant times (max size of confusion set)^2 times (max states per automaton)^2. (Crucially, there are only g choices for each of w_L and w_R.) Thus, the grammar constant S is typically small.

With careful coding it is possible to improve the runtime somewhat, from $O(n^3g^4t^4|M|)$ to $O(n^3g^3t^2m)$. Perhaps only some link types are possible in line 15, so $|M|$ can be replaced by m. We can save a factor of g at line 11, because the restriction on $sig(b).w_L$ means that we only need to iterate through S/g of the signatures. Finally, we can reduce S to just $O(g^2t)$, which helps both time and space complexity. Instead of Discover() using a single chart to maintain the best analysis with signature $(L?, R?, w_L, w_R, s_r, s_\ell, simple?)$, it will use two charts to maintain, respectively, the best analyses with signatures $(L?, R?, w_L, w_R, s_r, \text{HALTED}, simple?)$ and $(L?, R?, w_L, w_R, \text{HALTED}, s_\ell, simple?)$. Each of these two analyses has already had one stopweight added to its weight. The algorithm should now be modified to select a from the first chart and b from the second chart.

5 Conclusions

This paper has introduced a new formalism, weighted bilexical grammars, in which individual lexical items can have idiosyncratic selectional influences on each other. Such "bilexicalism" has been a theme of much current

[8] In the case where $start = 0$, also add $stopweight(sig(b).s_\ell)$; if $end = n + 1$, also add $stopweight(sig(a).s_r)$.

work. The new formalism can be used to describe bilexical approaches to both dependency and phrase-structure grammars, and its scoring approach is compatible with a wide variety of probability models.

The obvious parsing algorithm for bilexical grammars (used by most authors) takes time $\Theta(n^5)$. A more efficient $O(n^3)$ method is exhibited. The new algorithm has been implemented and used in a large parsing experiment [Eisner 1996b].

References

[Alshawi 1996] Hiyan Alshawi. Head automata for speech translation. *Proceedings of the fourth International Conference on Spoken Language Processing*, Philadelphia. cmp-lg/9607006.

[Charniak 1995] Eugene Charniak. Parsing with context-free grammars and word statistics. Technical Report CS-95-28, Dept. of Computer Science, Brown Univ.

[Church 1988] Kenneth W. Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the 2nd Conf. on Applied NLP*, 136–148, Austin, TX.

[Collins 1996] Michael J. Collins. A new statistical parser based on bigram lexical dependencies. *Proceedings of the 34th Annual ACL*, Santa Cruz, July, 184–191. cmp-lg/9605012.

[Collins 1997] Michael J. Collins. Three generative, lexicalised models for statistical parsing. *Proceedings of the 35th ACL and 8th European ACL*, Madrid, July. cmp-lg/9706022.

[Earley 1970] Earley, J. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM* 13(2): 94-102.

[Eisner 1996a] Jason M. Eisner. Three new probabilistic models for dependency parsing: an exploration. *Proceedings of COLING-96*, Copenhagen, August, 340–345. cmp-lg/9706003.

[Eisner 1996b] Jason M. Eisner. An empirical comparison of probability models for dependency grammar. Technical Report IRCS-96-11, IRCS, University of Pennsylvania. cmp-lg/9706004.

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

[Graham *et al.* 1980] Graham, S.L., Harrison, M.A. and Ruzzo, W.L. An Improved Context-Free Recognizer. *ACM Transactions on Prog. Languages and Systems* 2(3):415-463.

[Lafferty *et al.* 1992] John Lafferty, Daniel Sleator, and Davy Temperley. Grammatical trigrams: A probabilistic model of link grammar. In *Proc. of the AAAI Conf. on Probabilistic Approaches to Natural Language*, October.

[Mel'čuk1988] Igor A. Mel'čuk. *Dependency Syntax: Theory and Practice.* State University of New York Press.

[Milward 1994] David Milward. Dynamic dependency grammar. *Linguistics and Philosophy* **17**: 561–605. Netherlands: Kluwer.

[Pollard & Sag 1994] Carl Pollard & Ivan Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press.

[Resnik 1993] *Selection and Information: A Class-Based Approach to Lexical Relationships.* Ph.D. dissertation (Technical Report IRCS-93-42), University of Pennsylvania.

[Schabes *et al.* 1988] Yves Schabes, Anne Abeillé, & Aravind Joshi. Parsing strategies with 'lexicalized' grammars: Application to Tree Adjoining Grammars. Proceedings of COLING-88, Budapest, August.

[Wu 1995] Dekai Wu. An algorithm for simultaneously bracketing parallel texts by aligning words. *Proceedings of ACL-95*, MIT.