A Finite-State Parser with Dependency Structure Output

David Elworthy*
Canon Research Centre Europe Ltd.,
1 Occam Court
Occam Road
Guildford GU2 5YJ
dahe@acm.org

September 28, 1999

Abstract

Dependency parsers and finite-state parsers are both capable of rapid and robust parsing of natural language. Dependency parsers produce richer output structures, while finite-state parsers can be more efficient. We show how a finite-state parser can be used to produce dependency structures for most phrase types, with an $O(n^2)$ complexity in the number of words. The parser allows syntactically ambiguous structures to be packed into a single representation. The parser has been used as a component of a natural-language information retrieval system, operating in English and Japanese.

1 Dependency parsing and finite-state parsing

Recent work in parsing natural language has shown a shift away from the heavy-weight theories of syntax and semantics, towards grammar formalisms which can be parsed efficiently and which make it easy to write robust grammars. Many of the lightweight techniques use either finite-state parsing, or some variant of dependency grammars. Finite-state parsing offers the advantages of computational efficiency, and the ability to easily integrate a number of levels of processing, from the phonological (Bird and Ellison, 1994) through to the syntactic (Roche, 1997; Roche and Schabes, 1997). Dependency grammars have a long history in linguistics and NLP, from the linguistically inclined work of Mel'čuk (1988), to computation implementations such as the dependency parser of Järvinen and Tapanainen (1997). A related kind of grammar is the Link grammar of Sleator and Temperley (1991).

The processing complexity of dependency grammars can be as low as $O(n^3)$ (Eisner, 1996), where n is the number of words in the input. However, as Neuhaus and Bröker (1997) show, if certain sorts of linguistic construct are allowed, the processing complexity becomes exponential, up to being NP-complete. The specific constructs that cause this are non-projective ones, in which links between words must cross over, i.e. if there is a sequence of words $W_i ... W_j ... W_k$ in which W_i and W_k are linked, then W_j has a link to a word preceding W_i or following W_k. Neuhaus and Bröker show that such constructs can be found in English, in the form of topicalisation, and in German. Similar problems arise when the parser is allowed to tolerate ungrammatical input. For example, if W_i and W_k were spurious words in the input (say the

*Author's current address: Microsoft Research Ltd., St. George House, 1 Guildhall Street, Cambridge CB2 3NH, UK

output of a faulty speech recogniser), a link between them might force a grammatically correct link from W_j to make a crossing link.

Figure 1 gives an example of a dependency structure: the purely structural parse tree above has the dependency structure below, in which various different kinds of relationship between words are identified, such as nominal modification, specification by a determiner and agent realised as subject of an active verb. The dependencies point from the modifying item to the modified one[1].

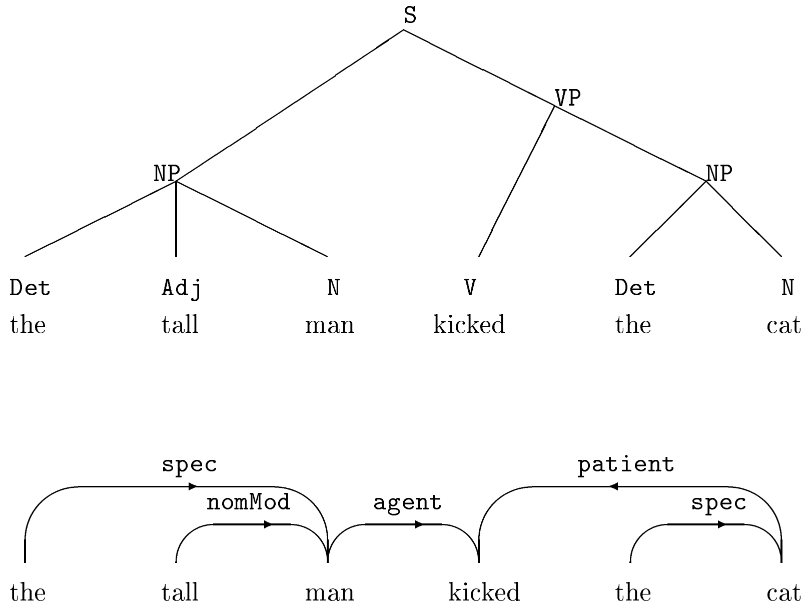


Figure 1: Phrase bracketing and dependency structure

By contrast, it is possible to write finite-state parsers which have a complexity of $O(n^2)$ even with ungrammatical input, provided the grammar can be determinised. To see this, assume that a left-to-right longest match parsing strategy such as that of Abney's SCOL parser (Abney, 1996) is being used. In this approach, the parser attempts to form a phrase from the longest prefix of the input which matches a rule in the grammar. If it succeeds, it continues with the remainder of the input. If not, the first word is "punted", i.e. it remains unparsed, and the parser resumes with the following word. The $O(n^2)$ complexity arises as follows: consider an input phrase in which the first $n-1$ words form the left prefix of a pattern in the grammar, but the nth word cannot complete the phrase. The parser will then skip the first word and attempt to parse starting with the second one. Again, words $2...n-1$ might form the prefix of a phrase, but the final word could result in the parse failing. In the worst case, the parser could have to try n passes through the input, successively trying n, $n-1$, $n-2$... words, resulting in an overall complexity of $O(n^2)$. Note that this analysis relies on the finite-state machines being deterministic; without it the complexity is at best $O(n^3)$.

Abney's parser, and some other finite-state parsers such as SRI's FASTUS (Hobbs *et al.* 1997), use a series of levels or cascades. Each level contains parse rules for one or more

[1] The relationships show here are the not the ones used in our grammar – they are just to give a flavour of what dependency structures look like

phrase types, and passes on any completed phrases and any unparsed elements between them to the next level. Thus, the input to the first level is a sequence of words and the output is a mixture of unparsed words and phrases of depth 1; at the second level, the output is a mixture of new phrases of depth 2 and words and phrases of depth 1 which could not be parsed; and so on. The number of levels is fixed in advance, and so the effect of cascading is to applying a constant to the complexity, but not to increase it overall. Ungrammaticalities and "difficult" constructs such as topicalisation can be handled by special rules in the grammar, the former mopping up unattached words and incorporating them into the structure in some arbitrary position, and the latter by special case rules in the later levels. These rules can be written to cover the special cases exactly by means of very specific finite-state patterns, making it easy to engineer the grammar to fit cases that cannot be handled in a more principled way.

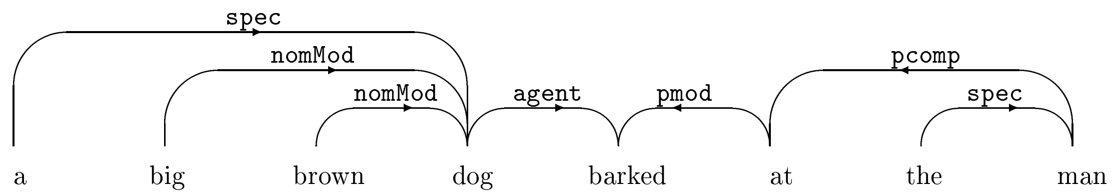
In spite of their advantages, one area where finite-state grammars show some weakness compared to dependency grammars is in the richness of the parse. Finite-state parsers typically construct a phrase bracketing to a fixed maximum depth (sometimes a depth of one, for non-cascaded parsers), which reflects the syntactic constituency of the input. By contrast, dependency parsers output modification relationships between words, which abstracts away from their linear grouping. Although dependency structures cannot be described as semantic, in the sense that they do not have a model theory or a proof theory, they do nevertheless give some representation of meaning, in that relationships between words which are not purely structural are be expressed in the dependency structure. An illustration of dependecy structures as a surrogate for a semantic representation can be found in Alshawi (1996).

In this paper, we will show how a finite-state parser can be modified to output dependency structures as well as a phrase bracketing. The computational efficiency and ease of writing grammars that we get from a finite-state parser remain, and we gain the richer output structure of a dependency structure. Furthermore, although the parser is deterministic, it is still possible to capture some kinds of ambiguity in a straighforward way, which deterministic finite-state parsers cannot normally do. The resulting parser has been used as a component of a natural language-based system for retrieval of captioned images.

2 A finite-state parser with dependency output

A simple technique can be used to make the finite-state parser produce dependency structures. As well as outputting the phrase bracketing, the parser also stores words "variables" in response to annotations included in the grammar rules. There are two sorts of variables: unindexed ones, which store lists of words, and indexed ones, which provide a mapping from words to lists of words. The indexed variables stand for dependency relations, and the unindexed ones provide pointers to non-dependent elements such as the overall head. Each variable has a name, which corresponds to the type of dependency relationship. An example is shown in figure 2, which shows a depdendency structure and its representation as a set of variables. All the variables are indexed, except *head*.

For example, the relationship between *dog* and *barked* is captured in the `agent` variable, which can be read as saying "the agent of *barked* is *dog*". A variable indexed on a particular word can point to several words, as in the example of `nomMod` (nominal modification), where both *big* and *brown* modify *dog*. Strictly speaking, the words should be assigned unique indices,




```
head           = barked
agent[barked] = dog
nomMod[dog]    = big, brown
spec[dog]      = a
pmod[barked]  = at
pcomp[at]      = man
spec[man]      = the
```

Figure 2: Dependency structure and variables

in case the same word-type appears as more than one word-token in the sentence.

2.1 Extracting the dependencies

The variable assignments are established by augment the rules of the finite-state grammar with "instructions". The kinds of instructions are introduced here, and below we will look at implementation issues. As in most finite-state parsers, the grammar rules are written as regular expressions using basic elements (lexical and phrasal categories) to which are applied alternation ($a|b$), repetition ($a*$ and $a+$), and concatenation ($a\ b$). The basic elements of the expressions are further annotated with the instructions. Each rule includes the phrasal category we are trying to construct. For implementation reasons, a list of the variables which will be set by the rule is included.

Instructions can take several forms. In the following, by default "variable" means a variable of the phrase type which the rule recognises.

- For a lexical category C and an unindexed variable v, *C:v* means that when a word with category C appears as part of the phrase, then it is stored in variable v.
- For a lexical category C, an unindexed variable v, and an indexed variable w, *C:w*$\{v\}$ means store the word with category C in variable w, indexed on the value of variable v.
- For a phrasal category P, an indexed or unindexed variable y belonging to P, and a variable w of the same sort as y, *P:w=y* means copy the value of y to w.
- For a phrasal category P, an unindexed variable x belonging to P, a variable of either sort y belonging to P, and an indexed variable w of the phrase we are parsing, *P:w[x]=y* means copy the value of y to w indexed on x.
- For a phrasal category P, an unindexed variable v belonging of the phrase we are parsing, a variable of either sort y belonging to P, and an indexed variable w of the phrase we are

parsing, $P{:}w\{v\}=y$ means copy the value of y to w indexed on v.

In addition, as a matter of notational convenience, the notation $P : \wedge$ can be used, meaning copy all the variables from a phrase of category P to the phrase we are recognising, for variables which exist in both categories and which are not otherwise copied. Note that is purely syntactic sugar, to make the rules easier to write.

Figure 3 is an example which illustrates some of the types of instruction. The rules can

```
NP(head, spec[], nomMod[])
     = Det:spec{head} (Adj:nomMod{head})* (N:nomMod{head})* N:head
VP(head, patient[], spec[], nomMod[])
     = V:head NP:(patient[head]=head,^)?
S(head, agent[], patient[], spec[], nomMod[])
     = NP:(agent[head]=head,^) VP:^
```

Figure 3: Sample grammar

be glossed as follows. The first rule identifies noun phrases made up of a determiner, optional adjectival and nominal modifiers and a final noun. The final noun is stored in the head variable, the determiner in the specifier (*spec*) of the head, and all the adjectives and nouns as modifiers of the head (*nomMod*). The verb phrase rule stores the verb in the head variable of the verb phrase. If a noun phrase object is present, if copies the head of the noun phrase to the patient variable indexed on the head of the verb phrase, thus making a dependency link between the verb and the head noun of the object. The $\wedge$ copies the *spec* and *mod* variables up to the verb phrase. Finally, the sentence rule similarly links in the subject noun phrase as the agent, while copying all of the other dependencies, and the head of the verb phrase, to the sentence's variables[2].

Note that there is a difference in style between these rules and the ones used by Abney (1996) or Roche (1997) concerning how phrase brackets are constructed. Abney and Roche (and others in the field) include them as output symbols in a finite state transducer. In the approach described here, if the phrase indicated on the left hand side of the rule is successfully recognised as a result of matching the pattern on the right hand side, then the phrase brackets are inserted, more in keeping with conventional CFG rules.

There are a few details to consider. First, variables store ordered lists of words, as in `nomMod` in the example above. The order of the words is the same as the order in the original phrase. If we index on a variable whose value is a list of words, then we effectively construct dependency links from each of these words to the target one. Secondly, the different kinds of instruction do not allow the indexing to be on anything other than an unindexed variable. For example, in $P{:}w[x]=y$, x must be unindexed. Complex indexing such as $P{:}w[x[y]] = z$ is not allowed. These restrictions were introduced in the interests of keeping the implementation simple, but they can also be seen as a linguistic restriction against descending into an existing structure and extracting information from within it.

[2] We are assuming, for the sake of this example, that the subject realises the agent role and the object the patient role

2.2 Implementation

There are two issues to consider in implementation. The first is how we set the variables. No variable must be set until a whole phrase has been recognised, and unindexed variables of the result phrase must be set before indexed ones, since the indexing operation may refer to these variables. This is easy to implement by storing the instructions on two agendas, one for unindexed variable and one for indexed ones. The agendas are executed when the whole phrase has been recognised to perform the variable settings.

A more difficult problem is that of determinisation. In order to obtain the $O(n^2)$ computational complexity, the finite-state machines derived from the grammar must be deterministic. There are standard algorithms for compiling regular expressions into non-deterministic finite-state machine, which can then be determinised (Aho *et al.*, 1986), provided the grammar actually is deterministic. However, once variables are included in the grammar, the usual determinisation algorithms break down. Figure 4 shows an example. The regular expression `Det (N:mod)* N:head` compiled into the regular expression in (a) which can then be determinised as (b) if the variables are ignored. However, once variables are added, the finite-state machine can no longer be determinised, as the instruction on the N arcs depend on whether they are being traversed for the last noun in the phrase (stored in `head`), or an earlier one (stored in `mod`). The problem is that the instruction to be applied cannot be determined by simply looking at the input symbol, in the way that the decision about which arc to follow can be. There can be multiple transitions out of a state which have the same input symbol but different instructions, and the transition to take can only be decided when next symbols of the input are known.

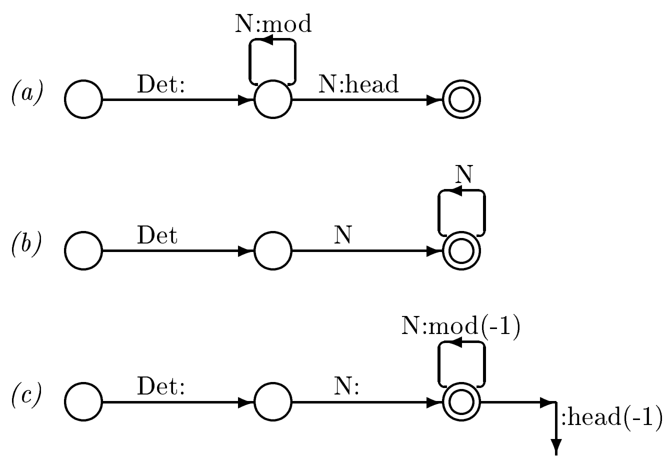


Figure 4: Determinisation of `Det (N:mod)* N:head`. (a) Compiled FSM (b) Simple determinisation (c) Determinisation with deferred transitions

Fortunately, the problem can be solved in most cases, using an algorithm described by Roche and Schabes (1995). Their algorithm is used for determinising finite-state tranducers, in which each arc is labelled with an input and an output symbol. If there are two arcs with the same input symbol, they are replaced with a single arc, and the output symbols are deferred to a later transition when it can be decided which one to apply. Sometimes this will be the next transition, but sometimes the output must be deferred by several steps through the finite-state network, accumulating other deferred outputs as it does so. If a state with no further transitions is reached in this process, all deferred outputs are placed on a transition with no input symbol. The results of this can be seen in figure 4 (c), where the indexes in *mod(-1)* and *head(-1)* show how many input symbols back the deferred output comes from. The arrow from the final state is a "mopping up" transition, which executes any deferred outputs that remain. One drawback of the Roche and Schabes algorithm is that if it is not possible to determinise the finite-state machine, then the algorithm fails to terminate. In practice, a check can be added so that if it has been round the determinisation loop more than a given number of times, an error message can be issued[3].

2.3 Ambiguity and packing

Determinising the finite-state machines buys computational efficiency, at the cost of producing a single output even if the input is ambiguous. The two main sources of ambiguity are lexical, such as alternate parts of speech for a word, and structural, such as prepositional phrase attachment. Lexical ambiguity can be handled by parsing with each possible sequence of parts of speech. To avoid a combinatorial explosion, a part of speech tagger can be used to find the most likely one. The error detection technique described in Elworthy (1997) can help here, in allowing a small number of ambiguities to remain in order to avoid tagger errors. Experience from an English parser shows that many of these ambiguities, for example between a noun and a present participle (e.g. *building*) are easily resolved in the grammar.

Structural ambiguity is a more serious problem for deterministic parsers, since picking one structure can often lead to the incorrect reading. Statistical information can sometimes be used to help resolve the ambiguity[4]. However, the use of variables to represent dependency allows an alternative scheme to be employed, in which the ambiguities are packed into a single structure. Although this may not solve all problems to do with structural ambiguity, two of the more common ones in English – PP attachment and noun compounding – can easily be handled this way.

To construct packed dependency structures, we place the range (value) of more than one indexed variable, so that it can be accessed from more than one word. One way of doing this is shown in the following grammar, in which the same word is initially stored in the range of an indexed variable and an unindexed one, and the unindexed value is transferred to an indexed one when we are ready to construct the second link. The grammar in figure 5 shows how this is done for prepositional phrase attachment ambiguity. The first PP rule builds simple prepositional phrases, which can then either be built into larger prepositional phrases or combined with a noun phrase. The variable *phead* provides access to the head of the NP within the PP, even though the preposition is taken as the overall head of the PP, while em pcomp provides access via the preposition. In the second PP rule, the existing variabled are copied up

[3] Roche and Schabes's paper in fact includes a slight error, in that they claim their algorithm "does not terminate when run on a nonsubsequential function", i.e. one for which no determinisation exists. Later in the paper, they prove the statement that "completeness: the algorithm terminates in the case of subsequential functions" which does not amount to the same thing. An example of a nonsubsequential transducer for which which their algorithm incorrectly terminates is `(a:x)|(a:y)`. Fortunately cases such as this can be easily detected in the implementation.

[4] See Ratnaparkhi (1998) for one technique and references to a number of others

```
NP(head)
     = Det N:head
PP(prep, pcomp[], phead)
     = Prep:prep NP:(pcomp[prep]=head, phead=head)
PP(prep, pcomp[], phead)
     = PP:^ PP:(^,pmod{phead}=prep)
NP(head, pmod[], pcomp[])
     = NP:^ (PP:pmod{head}=prep,pcomp=pcomp,pmod=pmod)*
```

Figure 5: Grammar for packing

(`P:^`), including *phead*, and a *pmod* link from this *phead* is then made to the preposition of the second PP (`pmod{phead}=prep`). When the phrase *the man in the park on a hill* is parsed, the result is as shown in figure 6, with *on a hill* attached in the final structure to both *man* and *park*. A similar approach can be used for noun compounds.

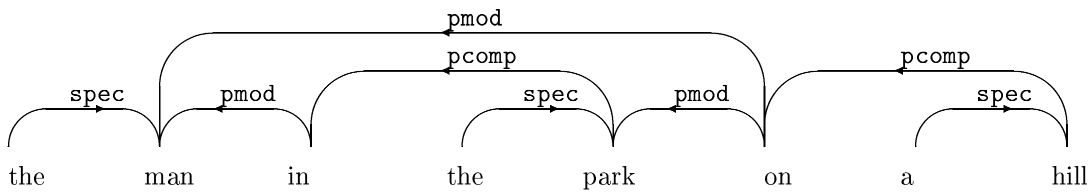


Figure 6: Packed structure

2.4 Applications

The finite-state dependency parser described here has been implemented in C++ as part of an information retrieval system called ANVIL. The grammar is compiled and determinised off-line. Within ANVIL, it is used to parse user queries and phrases which annotate objects in a database. An error-tolerant matching algorithm then assigns a similarity score between the query and the annotation by exploring the dependency structures. Initial evaluation suggest that better retrieval results can be achieved compared with conventional keyword techniques.

Two grammars have been written for this system. The first covers a wide range of English noun phrases, and has 13 levels and 17 rules, each producing a single finite-state machine. Note that these figures are a bit artificial, as some levels contain only a single rule, and some rules contain several disjunctions. The grammar compiles into determinised finite-state machines with a total of 272 states. The second is a wide coverage grammar of Japanese, with 26 levels and 370 rules, compiling into 9276 states. The Japanese grammar currently takes a lot of memory when loaded into the parser, although this can be reduced by using a simple system of feature to combine similar rules.

3 Discussion and Conclusions

We now briefly look at two pieces of work which have some similarities with the one described here. The first is Abney's SCOL parser, which has already been referred to. As noted, SCOL inserts phrase brackets by including them as output symbols of a finite-state transducer. Simple variables can be set in the same way. For example, to mark the head, Abney allows an output symbol of the form $h=$, which is subsequently interpreted as an instruction to set a variable. There is no mechanism for indexing, and Abney does not mention the problem of determinising finite-state machines with output symbols, which was solved here by the use of Roche and Schabes's algorithm.

A second comparison is with the finite-state dependency parser of Oflazer (1999). In this approach, dependency links are constructed by means of transducers which introduce "channels", representing potential locations for dependency links. The actual links are then constructed if the lexical items and morphemes are specified to have suitable potential connections, and constraints such as projectivity are respected. There are some advantages to this approach, compared to ours: the potential to form links and the constraints are expressed as separate finite state transducers which can be composed into the overall transducer. As a result, the formalism is more modular; for example, constraints can be selected as appropriate for the application. It is also declarative and reversible, so that the transducer can take either the surface form or the dependency structure as its input. In general, this is more in keeping with the prevailing style of finite-state processing in the field, as described in a number of the papers in Roche and Schabes (1997), for example. No analysis of the processing complexity is given. The formalism is not very perspicuous, in that it can be quite difficult to see how the links between items will be formed.

As mentioned, the finite-state dependency parser has proved very useful in the ANVIL retrieval system. It allows us to implement a phrase matching algorithm which respects the relationships between the words as well as matching the words themselves. We can therefor avoid false matches between, say, *camera with a lens* and *camera lens*. Equally, the finite-state dependency parser avoids problems which arise from a full semantic representation which can be hard to compute for unconstrained input, and which can be over-specified for the needs of the application, so adding extra ambiguity and processing cost during retrieval.

Acknowledgements

The parser was implemented by the author and Aaron Kotcheff, and the work also benefitted from discussions with Tony Rose and Amanda Clare.

References

Abney, Steven (1996). Partial Parsing via Finite-State Cascades. In *Proceedings of the ESSLLI '96 Robust Parsing Workshop*. Also available from http://www.sfs.nphil.uni-tuebingen.de/~abney/.

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Alshawi, Hiyan (1996). Head Automata and Bilingual Tiling: Translation with Minimal Representations. In *Proceedings of the 34th ACL*, pp. 167–176.

Bird, Steven and T. Mark Ellison (1994). One-Level Phonology: Autosegmental Representations and Rules as Finite Automata. *Computational Linguistics*, volume 20, number 1, pp. 55–90.

Eisner, Jason M. (1996). Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of COLING-96*, pp. 340–345. cmp-lg/9706003.

Elworthy, David (1997). Automatic Error Detection in Part-of-speech Tagging. In Jones and Somers (1997), pp. 256–268. Reprinted from *Proceedings of the Conference on New Methods in Language Processing, 1994*.

Hobbs, Jerry R., Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel and Mabry Tyson (1997). FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text. In Roche and Schabes (1997), pp. 383–406. cmp-lg/9705013.

Järvinen, Timo and Pasi Tapanainen (1997). A Dependency Parser for English. Technical Report TR-1, Department of General Linguistics, University of Helsinki.

Jones, Daniel and Harold Somers, eds. (1997). *New Methods in Language Processing.* UCL Press.

Mel'čuk, Igor A. (1988). *Dependency Syntax: Theory and Practice.* SUNY Press.

Neuhaus, Peter and Norbert Bröker (1997). The Complexity of Recognition of Linguistically Adequate Dependency Grammars. In *Proceedings of the 35th ACL and 8th EACL*, pp. 337–343.

Oflazer, Kemal (1999). Dependency Parsing with an Extended Finite State Approach. In *Proceedings of the 37th ACL.*

Ratnaparkhi, Adwait (1998). Statistical Models for Unsupervised Prepositional Phrase Attachment. In *Proceedings of the 36th ACL and 17th COLING*, pp. 1079–1085.

Roche, Emmanuel (1997). Parsing with Finite-State Transducers. In Roche and Schabes (1997), pp. 241–281.

Roche, Emmanuel and Yves Schabes (1995). Deterministic Part-of-Speech Tagging with Finite-State Transducers. *Computational Linguistics*, volume 21, number 2, pp. 227–253. Also in Roche and Schabes (1997).

Roche, Emmanuel and Yves Schabes, eds. (1997). *Finite State Language Processing.* MIT Press.

Sleator, Daniel D. K. and Davy Temperley (1991). Parsing English with a Link Grammar. Technical report CMU-CS-91-196, School of Computer Science, Carnegie Mellon University. cmp-lg/9508004.