A Fundamental Algorithm for Dependency Parsing

Michael A. Covington

Artificial Intelligence Center
The University of Georgia
Athens, GA 30602-7415 U.S.A.
mc@uga.edu

Abstract– This paper presents a fundamental algorithm for parsing natural language sentences into dependency trees. Unlike phrase-structure (constituency) parsers, this algorithm operates one word at a time, attaching each word as soon as it can be attached, corresponding to properties claimed for the parser in the human brain. Like phrase-structure parsing, its worst-case complexity is $O(n^3)$, but in human language, the worst case occurs only for small n.

1   Overview.

This paper develops, from first principles, several variations on a fundamental algorithm for parsing natural language into dependency trees. This is an exposition of an algorithm that has been known, in some form, since the 1960s but is not presented systematically in the extant literature.

Unlike phrase-structure (constituency) parsers, this algorithm operates one word at a time, attaching each word as soon as it can be attached. There is good evidence that the parsing process used by the human mind has these properties [1].

2   Dependency grammar.

2.1   The key concept.

There are two ways to describe sentence structure in natural language: by breaking up the sentence into **constituents** (phrases), which are then broken into smaller constituents (Fig. 1), or by drawing links connecting
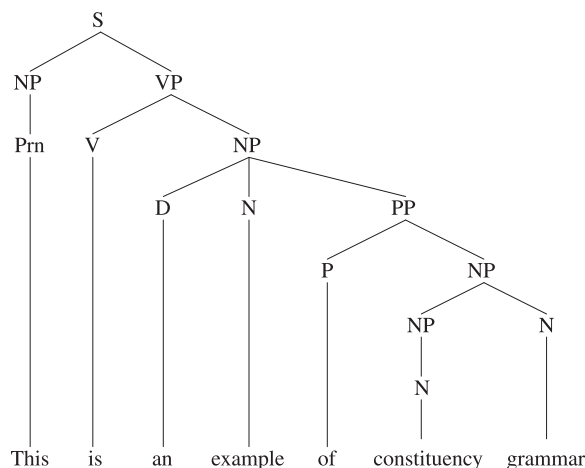
Figure 1: A constituency tree.

individual words (Figs. 2, 3). These are called **constituency grammar** and **dependency grammar** respectively.

Constituency grammar appears to have been invented only once, by the ancient Stoics [12], from whom it was passed through formal logic to linguists such as Leonard Bloomfield, Rulon Wells, Zellig Harris, and Noam Chomsky. It is also the basis of formal language theory as studied by computer scientists.

Dependency grammar, on the other hand, has apparently been invented many times and in many places. The concept of a word-to-word link occurs naturally to any grammarian who wants to explain agreement, case assignment, or any semantic relation between words. Dependency concepts are found in traditional Latin, Arabic, and Sanskrit grammar, among others. Computer implementations of dependency grammar have attracted interest for at least 40 years [9, 7, 8, 4, 5, 13], but there has been little systematic study of dependency parsing,

Figure 2: A dependency tree is a set of links connecting heads to dependents.
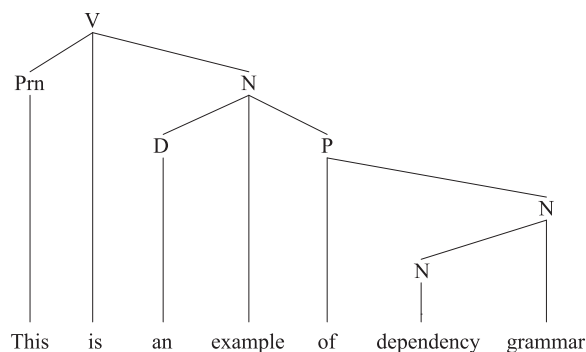

Figure 3: This representation of a dependency tree preserves the word order while depicting the tree structure plainly. To get from a head to its dependents, go downhill.

appparently due to the widespread misconception that all dependency parsers are notational variants of constituency parsers.

2.2 Dependency trees.

Whenever two words are connected by a dependency relation, we say that one of them is the **head** and the other is the **dependent**, and that there is a **link** connecting them. In general, the dependent is the modifier, object, or complement; the head plays the larger role in determining the behavior of the pair. The dependent presupposes the presence of the head; the head may require the presence of the dependent.

Figure 2 shows the dependency structure of a sentence. Essentially, a dependency link is an arrow pointing from head to dependent. The dependency structure is a tree (directed acyclic graph) with the main verb as its root (head).

Figure 3 shows a way to display the word order and the tree structure at once. To get from a word to its dependents in this kind of diagram, go downhill.

In what follows, a dependent that precedes its head is called a **predependent**; one that follows its head, a **postdependent**.

I shall say that a word is **independent** (headless) if it is not a dependent of any other word.

Note that in the dependency tree, constituents (phrases) still exist. Any word and all its dependents, their dependents, etc., form a phrase. I shall say that dependents, dependents of dependents, etc., are **subordinate** to the original word, which in turn **dominates** (is **superior** to) them.

A word **comprises** itself and all the words that it dominates. That is, the head of a phrase comprises the whole phrase.

2.3 Generative power.

In 1965, Gaifman [6] proved that dependency grammar and constituency grammar are **strongly equivalent** — that they can generate the same sentences and make the same structural claims about them — provided the constituency grammar is restricted in a particular way. The restriction is that one word in each phrase is designated its **head**, and the phrase has no name or designation apart from the designation of its head.

That is tantamount to saying that a noun phrase has to be built around a noun, a verb phrase around a verb, and so forth. Just take the category name "noun" or "verb," add "phrase," and you have the name of the phrase that it heads. (In a pure constituency grammar, NP and VP are atomic symbols not related to N and V, a fact all too seldom appreciated.)

Linguists have accepted this proposed restriction for other reasons; they call it **X-bar theory** [11]. Thus, constituency grammar as currently practiced is very close to being a notational variant of dependency grammar. Figure 4 shows interconversion of dependency and constituency trees. A bar over a category label indicates
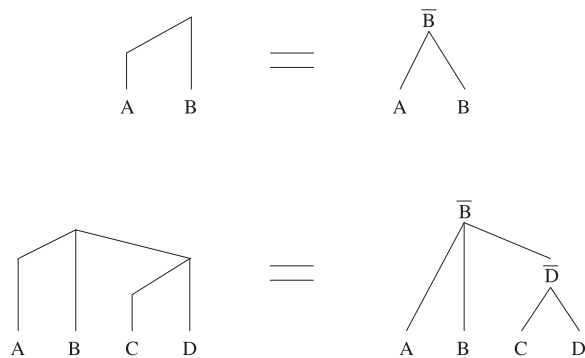
Figure 4: Equivalent dependency and constituency trees.

that it labels a phrase rather than a word.

2.4 The appeal of dependency parsing.

In what follows I shall explore some parsing algorithms that use the dependency representation. Please note that I am not claiming any significant difference in generative power between dependency grammar and constituency grammar; still less am I claiming that English, or any other human language, "is a dependency language" rather than a constituency language, whatever that might mean. Nor do I address any technical aspects of constructing an adequate dependency grammar of English. My concern is only the formalism. *Prima facie,* dependency parsing offers some advantages:

- Dependency links are close to the semantic relationships needed for the next stage of interpretation; it is not necessary to "read off" head-modifier or head-complement relations from a tree that does not show them directly.

- The dependency tree contains one node per word. Because the parser's job is only to connect existing nodes, not to postulate new ones, the task of parsing is in some sense more straightforward. (We will presently see that the actual order of complexity is no lower, but the task is nonetheless easier to manage.)

- Dependency parsing lends itself to word-at-a-time operation, i.e., parsing by accepting and attaching words one at a time rather than by waiting for complete phrases.

  Abney [1] cites several kinds of evidence that the parser in the human mind operates this way. Consider for example a verb phrase that may or may not contain a direct object, such as *sang loudly* (vs. *sang*

songs loudly). A top-down constituency parser has to choose *a priori* whether to expect the object or not, before it has any way to know which choice is right, and then has to backtrack if it guessed wrong; that is *spurious local ambiguity,* apparently absent in human parsing. A bottom-up constituency parser cannot construct the verb phrase until all the words in it have been encountered; yet people clearly begin to understand verb phrases before they are over. My dependency parser has neither problem; it accepts words and attaches them with correct grammatical relations as soon as they are encountered, without making any presumptions in advance.

3 The parsing task.

The task of a dependency parser is to take a string of words and impose on it the appropriate set of dependency links. In what follows I shall make several assumptions about how this is to be done.

3.1 Basic assumptions.

- **Unity**: The end product of the parsing process is a single tree (with a unique root) comprising all the words in the input string.

- **Uniqueness**: Each word has only one head; that is, the dependency links do indeed form a tree rather than some other kind of graph.

  Most dependency grammars assume uniqueness, but that of Hudson [10] does not; Hudson uses multiple heads to account for transformational phenomena, where a single word has connections to more than one position in the sentence.

- **Projectivity (adjacency)**: If word A depends on word B, then all words between A and B are also subordinate to B. This is equivalent to "no crossing branches" in a constituency tree.

  Some dependency grammars assume projectivity, and others do not. In an earlier paper [3] I showed how to adapt dependency parsing to a language with totally free word order. This of course entails abandoning projectivity.

- **Word-at-a-time operation**: The parser examines words one at a time, attaching them to the tree as they are encountered, rather than waiting for complete phrases.

  This excludes dependency parsers that are simple notational variants of constituency parsers.

- **Single left-right pass**: Unless forced to backtrack because of ambiguity, the parser makes a single left-to-right pass through the input string. This is a vague requirement until the other requirements are spelled out more fully, but it excludes parsers that look ahead an indefinite distance, find a head, and back up to find its predependents (compare [14]).

- **Eagerness**: The parser establishes each link as early in its left-right pass as possible. Abney argues convincingly that eagerness is a property of the parsers in our heads [1].

3.2   Simplifying assumptions.

For this initial investigation I will make four more assumptions that will definitely need to be relaxed when parsing natural language with actual grammars.

- **Instant grammar**: I assume that the grammar can tell the parser, in constant time, whether any given pair of words can be linked, and if so, which is the head and which is the dependent. (In a real grammar, some links could be harder to work out than others.)

- **No ambiguity**: I assume that there is neither local nor global ambiguity in any parse tree; that is, every link put in place by the parser is part of the ultimately correct parse.

  This is clearly false for natural language, but by assuming it, I can postpone consideration of how to manage ambiguity. Psychological evidence indicates that the parsers in our heads encounter relatively little local ambiguity, and that they backtrack when necessary [1].

- **No inaudibilia**: The grammar does not postulate any inaudible elements such as null determiners, null auxiliaries, or traces. (Bottom-up parsers cannot respond to inaudibilia.)

- **Atomicity**: I assume that words are unanalyzable elements and that there are no operations on features or words' internal structure.

  Figures 5 and 6 show "test suites" of projective and non-projective structures that parsers should handle.

4   The obvious parsing strategy.

Given these assumptions, one parsing strategy is obvious. I call it a strategy and not an algorithm because it is not yet fully specified:
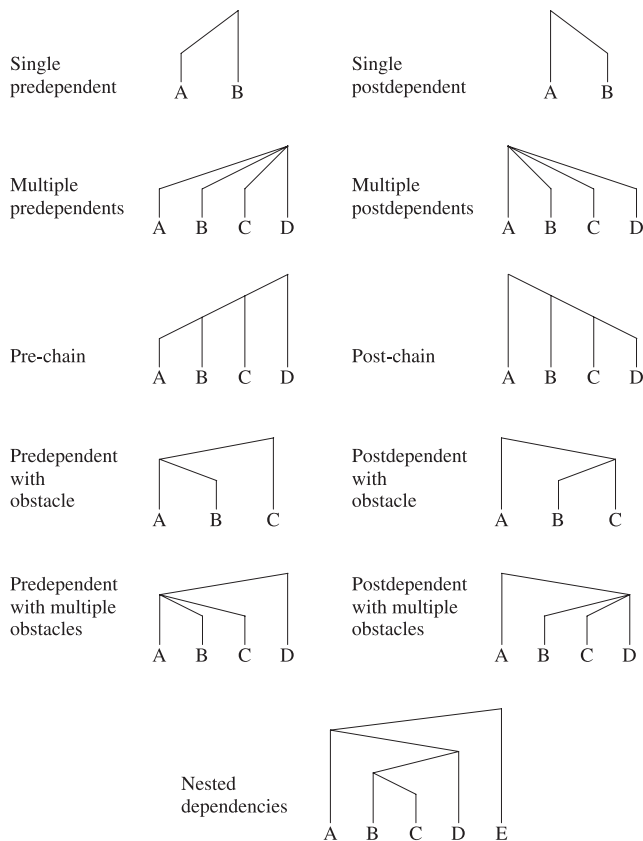

Figure 5: Some projective (non-crossing) structures that any dependency parser should handle.
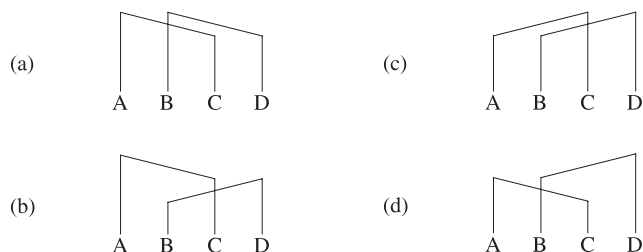

Figure 6: Some non-projective structures, allowed in some languages and not in others.

Strategy 1 (Brute-force search) *Examine each pair of words in the entire sentence, linking them as head-to-dependent or dependent-to-head if the grammar permits.*

That is, for n words, try all $n(n-1)$ pairs. Note that the number of pairs, and hence the parsing complexity, is $O(n^2)$. If backtracking were permitted, it would be $O(n^3)$, just like constituency parsing, because in the theoretical worst case, the whole process might have to be done afresh after accepting each word.

Implemented as a single left-to-right pass, the brute-force search strategy is essentially this:

Strategy 2 (Exhaustive left-to-right search)
Accept words one by one starting at the beginning of the sentence, and try linking each word as head or dependent of every previous word.

This still leaves the order of comparisons unspecified. When looking for potential links to word n, do we work backward, through words $n-1$, $n-2$, etc., down to 1, or forward, from word 1 to word 2 up to $n-1$?

Clearly, if the grammar enforces projectivity, or even if projective structures are merely predominant, then the head and dependents of any given word are more likely to be near it than far away. Thus, they will be found earlier by working backward than by working forward.

Whether it is better to look for heads and then dependents, or dependents and then heads, or both concurrently, cannot yet be determined. Thus we have two fully specified algorithms:

Algorithm ESH
(Exhaustive left-to-right search, heads first)
Given an n-word sentence:
[1] **for** $i := 1$ **to** n **do**
[2] **begin**
[3]   **for** $j := i - 1$ **down to** 1 **do**
[4]   **begin**
[5]   If the grammar permits,
              link word j as head of word i;
[6]   If the grammar permits,
              link word j as dependent of word i
[7]   **end**
[8] **end**

Algorithm ESD
(Exhaustive left-to-right search, dependents first)
Same, but with steps [5] *and* [6] *swapped.*

Note that although these algorithms are expressed in terms of arrays indexed by i and j, they can also be implemented with linked lists or in some other way.

5   Refining the algorithms.

Those naïve algorithms are obviously inefficient. A better dependency parsing algorithm should not even try links that would violate unity, uniqueness, or (when required by the language) projectivity.

Because the parser operates one word at a time, unity can only be checked at the end of the whole process: did it produce a tree with a single root that comprises all of the words? Uniqueness and projectivity, however, can and should be built into the parsing algorithm. Here is how to handle uniqueness:

Strategy 3 (Enforcing uniqueness)

- *Principle: When a word has a head, it cannot have another one.*

- *Implementation:*

    - *When looking for dependents of the current word, do not consider words that are already dependents of something else.*

    - *When looking for the head of the current word, stop after finding one head; there will not be another.*

This leads immediately to:

Algorithm ESHU
(Exhaustive search, heads first, with uniqueness)
Given an n-word sentence:
[1] **for** $i := 1$ **to** n **do**
[2] **begin**
[3]   **for** $j := i - 1$ **down to** 1 **do**
[4]   **begin**
[5]   If no word has been
                linked as head of word i, then
[6]   if the grammar permits,
              link word j as head of word i;
[7]   If word j is not a dependent
                of some other word, then
[8]   if the grammar permits,
              link word j as dependent of word i
[9]   **end**
[10] **end**

Algorithm ESDU
(Exhaustive search, dependents first, with uniqueness)
Same, but with [5–6] *and* [7–8] *swapped.*

Here the advantages of a list-based representation begin to become apparent. Rather than work through an array and perform tests to determine which elements

to skip, it is simpler to work through lists from which the ineligible elements have already been removed.

Here is an algorithm that works with two lists, *Wordlist* and *Headlist*, containing, respectively, all the words encountered so far and all the words that lack heads. Both lists are built by adding elements at the beginning, so they contain words in the opposite of the order in which they were encountered. As a result, searching each list from the beginning retrieves the most recent words first.

Algorithm LSU
(List-based search with uniqueness)
Given a list of words to be parsed,
and two working lists Headlist and Wordlist:

(Initialize)
$Headlist := []$;   (Words that do not yet have heads)
$Wordlist := []$;   (All words encountered so far)

repeat

  (Accept a word and add it to *Wordlist*)
  $W :=$ *the next word to be parsed*;
  $Wordlist := W + Wordlist$;

  (Dependents of W can only be in *Headlist*)
  for $D :=$ *each element of Headlist,*
        starting with the first
    begin
      if D *can depend on* W **then**
        begin
          link D *as dependent of* W;
          delete D *from Headlist*
        end
    end;
  (Look for the head of W; there can only be one)
  for $H :=$ *each element of Wordlist,*
        starting with the first
    if W *can depend on* H **then**
      begin
        link W *as dependent of* H;
        terminate this **for** *loop*
      end;
    if *no head for* W *was found* **then**
      $Headlist := W + Headlist$;

until *all words have been parsed.*

This time dependents are sought before seeking heads. The reason is that W, the current word, is itself added to *Headlist* if it has no head, and a step is saved by not doing this until the search of *Headlist* for potential dependents of W is complete. This is essentially the algorithm of my earlier paper [3].

6 Projectivity.

6.1 Definition.

Projectivity is informally defined as "no crossing branches." More formally:

- A tree is projective if and only if every word in it comprises a continuous substring.

- A word comprises a continuous substring if and only if, given any two words that it comprises, it also comprises all the words between them.

The second clause of this is simply the definition of "continuous" – a continuous substring is one such that everything between any of its elements is also part of it.

6.2 Building projectivity into the parser.

Now how does all of this apply to parsing? To build projectivity into a bottom-up dependency parser, we need to constrain it as follows:

(a) Do not skip a potential predependent of W. That is, either attach every consecutive preceding word that is still independent, or stop searching.

(b) When searching for the head of W, consider only the previous word, its head, that word's head, and so on to the root of the tree.

Constraint (b) is easy to understand. It says that if the head of W (call it H) precedes W, it must also comprise the word immediately preceding W; thus it is reachable by climbing the tree from that word. This follows from the definition of projectivity: the substring $H \ldots W$ must be continuous.

Constraint (a) says that the predependents of W are a continuous string of the words that are still independent at the time W is encountered.

Consider the words that, at any stage, still do not have heads, i.e., the contents of *Headlist* in the list-based parsing algorithm. Each such word is the head of a constituent, i.e., a continuous substring. That is, each still-independent word stands for the string of words that it comprises. The goal of the parser is to assemble zero or more of these strings into a continuous string that ends with W. Clearly, if any element is skipped, the resulting string cannot be continuous. Q.E.D.

Here is the list-based parsing algorithm with projectivity added. This algorithm was mentioned briefly in [3].

Algorithm LSUP
(List-based search with uniqueness and projectivity)

Given a list of words to be parsed,
and two working lists Headlist and Wordlist:

(Initialize)
Headlist := [];   *(Words that do not yet have heads)*
Wordlist := [];   *(All words encountered so far)*

repeat

(Accept a word and add it to Wordlist)
W := the next word to be parsed;
Wordlist := W + Wordlist;

(Look for dependents of W; they can only be
  consecutive elements of Headlist
  starting with the most recently added)
for *D := each element of Headlist,*
        starting with the first
  begin
    if *D can depend on W* **then**
      begin
        link D as dependent of W;
        delete D from Headlist
      end
    else
      terminate this **for** *loop*
  end;

(Look for the head of W; it must
        comprise the word preceding W)
H := the word immediately preceding W
        in the input string;
loop
  if *W can depend on H* **then**
    begin
      link W as dependent of H;
      terminate the loop
    end;
  if *H is independent* **then** *terminate the loop;*
  H := the head of H
end loop;
if *no head for W was found* **then**
  Headlist := W + Headlist;

until *all words have been parsed.*

7   Complexity.

We saw already that the complexity of the initial, brute-force search algorithm, with a completely deterministic grammar, is $O(n^2)$ because the search involves $n(n-1)$ pairs of words, and $n(n-1)$ approaches n^2 as n becomes large.

So far I have not introduced any mechanism for handling local ambiguity. The obvious way to do so is to **backtrack** – that is, return to the most recent untried alternative whenever an alernative is needed. If
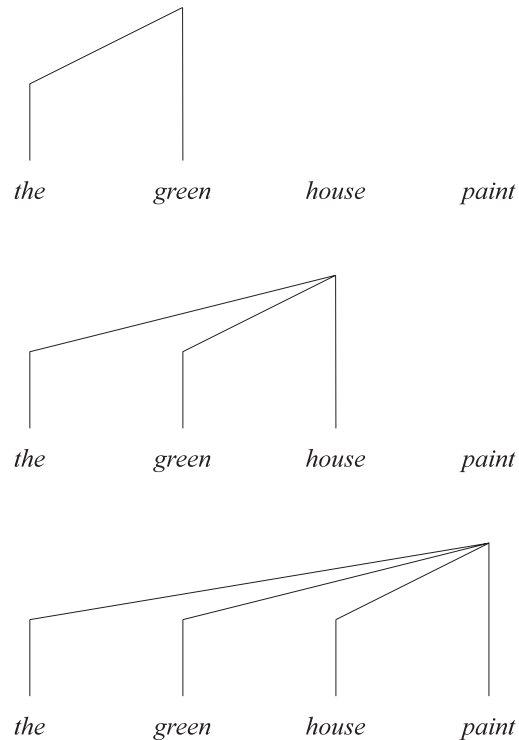

Figure 7: An instance of worst-case parsing complexity: after accepting each word, the parser has to rework the entire structure.

the parser is implemented in Prolog, backtracking is provided automatically.

The complexity of brute-force-search parsing with backtracking is $O(n^3)$ because, after each of the n words is accepted, the whole $O(n)$ process may have to be done over from the beginning. $O(n^3)$ is also the complexity of recursive-descent constituency parsing.

These complexity results are not affected by constraints to enforce unity and projectivity, since there are cases in which these constraints do not shorten the parsing process. Consider for example the local ambiguity in the phrase *the green house paint*. Not only is *the green* a valid phrase (as in "you forgot the green," said to a painter), but so are *the green house* and *the green house paint*. Thus, the parser must backtrack on accepting each successive word (Fig. 7).

At this point I am still assuming atomicity. Barton, Berwick and Ristad [2] prove that when lexical ambiguity and agreement features are present — that is, when words can be ambiguous and can be labeled with attributes — natural language parsing is NP-complete.

Bear in mind that these are worst-case results. An

important principle of linguistics seems to be that **the worst case does not occur,** i.e., people do not actually utter sentences that put any reasonable parsing algorithm into a worst-case situation. Human language does not use unconstrained phrase-structure or dependency grammar; it is constrained in ways that are still being discovered.

References

[1] Abney, Steven P. (1989) A computational model of human parsing. *Journal of Psycholinguistic Research* 18:129–144.

[2] Barton, G. Edward, Jr.; Berwick, Robert C.; and Ristad, Eric Sven (1987) *Computational Complexity and Natural Language.* Cambridge, Mass.: MIT Press.

[3] Covington, Michael A. (1990) Parsing discontinuous constituents with dependency grammar. *Computational Linguistics* 16:234–236.

[4] Fraser, Norman M. (1993) *Dependency Parsing.* Thesis, Ph.D., University of London.

[5] Fraser, N[orman] M. (1994) Dependency grammar. In *The Encyclopedia of Language and Linguistics,* ed. R. E. Asher, vol. 2, 860–864.

[6] Gaifman, Haim (1965) Dependency systems and phrase-structure systems. *Information and Control* 8:304–307.

[7] Hays, David G. (1964) Dependency theory: a formalism and some observations. *Language* 40:511–525.

[8] Hays, David G. (1966) Parsing. In David G. Hays, ed., *Readings in Automatic Language Processing.* New York: American Elsevier.

[9] Hays, D[avid] G., and Ziehe, T. W. (1960) *Studies in Machine Translation – 10: Russian Sentence-Structure Determination.* Research Memorandum RM-2538, The RAND Corporation, Santa Monica, California.

[10] Hudson, Richard A. (1991) *English Word Grammar.* Oxford: Blackwell, 1991.

[11] Jackendoff, Ray (1977) $\overline{X}$ *Syntax.* Cambridge, Mass.: MIT Press.

[12] Mates, Benson (1961) *Stoic Logic.* Berkeley: University of California Press.

[13] Sgall, Peter (1994) Dependency-based formal description of language. In *The Encyclopedia of Language and Linguistics,* ed. R. E. Asher, vol. 2, 867–872. Oxford: Pergamon Press.

[14] van Noord, Gertjan (1997) An efficient implementation of the head-corner parser. *Computational Linguistics* 23:425–456.