Research Report AI–1990–01

A Dependency Parser for Variable-Word-Order Languages

Michael A. Covington

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602

A Dependency Parser for Variable-Word-Order Languages

Michael A. Covington

Artificial Intelligence Programs
The University of Georgia
Athens, Georgia 30602

January 2, 1990

---

1 Introduction

This paper presents a new approach to the recognition of sentence structure by computer in human languages that have variable word order. In a sense, the algorithm is not new; there is good evidence that it was known 700 years ago (Covington 1984). But it has not been implemented on computers, and the modern implementations that are most like it fail to realize its crucial advantage for dealing with variable word order.[1] In fact, present-day parsing technology is so tied to the fixed word order of English that researchers in Germany and Japan customarily build parsers for English rather than their own languages.

The new algorithm uses dependency grammar. Unlike the more usual phrase structure grammars, a dependency grammar does not divide the sentence up into phrases (constituents); instead, it identifies the grammatical relations that connect one word to another. This is

[1]Phyllis McLanahan provided invaluable assistance with Russian data. The early stages of this work were supported by National Science Foundation Grant IST–85–02477. The VM/Prolog implementation of GULP was developed while visiting the Seminar für natfürlich-sprachliche Systeme, University of Tfübingen. Norman Fraser and Richard Hudson provided valuable encouragement during the later stages of the project. Responsibility for opinions and errors rests solely with the author.

advantageous in languages where the order of words is variable and many of the constituents are discontinuous.

The algorithm presented here is implemented in Prolog on an IBM 3090 and has been used successfully to parse Russian and Latin sentences. The IBM 3090 is well suited for this and other applications in artificial intelligence and symbolic computing because of its large address space, memory caching, and ability to prefetch along both alternatives of a conditional branch. Performance is currently limited not by the hardware, but by the VM/Prolog interpreter, which could be replaced by a considerably faster compiler.

2 Variable word order: the problem

2.1 Most human languages have partly variable word order

Most of the languages of the world allow considerably more variation of word order than does English. For example, the English sentence

The dog sees the cat.

has six equally grammatical Russian translations:

Sobaka vidit koshku.

Sobaka koshku vidit.

Vidit sobaka koshku.

Vidit koshku sobaka.

Koshku vidit sobaka.

Koshku sobaka vidit.

These differ somewhat in emphasis but not in truth conditions. The subject and object are identified, not by their positions, but by their inflectional endings (*-a* for nominative case and *-u* for accusative). By switching the endings, one can say "the cat sees the dog" without changing the word order:

Sobaku vidit koshka. (etc.)

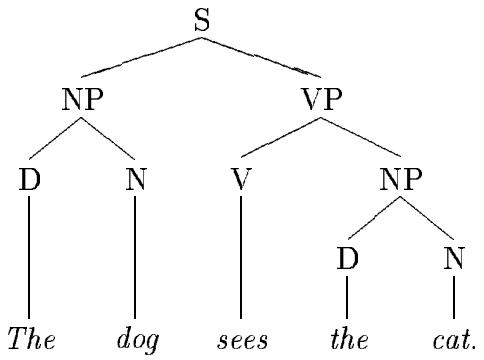
The languages of the world can be ranked as to the amount of word order variability they allow. For example:

Almost no variation:	Chinese, English, French
Some variation:	Japanese, German, Finnish
Extensive variation:	Russian, Latin, Korean
Maximum variation:	Walbiri (Australia)

Because Englih is at one end of the scale — permitting almost no word order variation — it is a poor sample of a human language to do research on. A priori, one would expect that parsing techniques developed solely for English might not work at all for other languages, and might not be correct on a deeper level even for English. In what follows I shall argue implicitly that this is the case.

2.2 Phrase-structure grammar (PSG) cannot handle variable word order

Virtually all present-day linguistic theories analyze the sentence by breaking it into substrings (constituents). For example:



Here *the dog* is a noun phrase, *sees the cat* is a verb phrase, and the grammar consists of phrase-structure rules (PS-rules) such as

$S \rightarrow NP + VP$

$NP \rightarrow D + N$

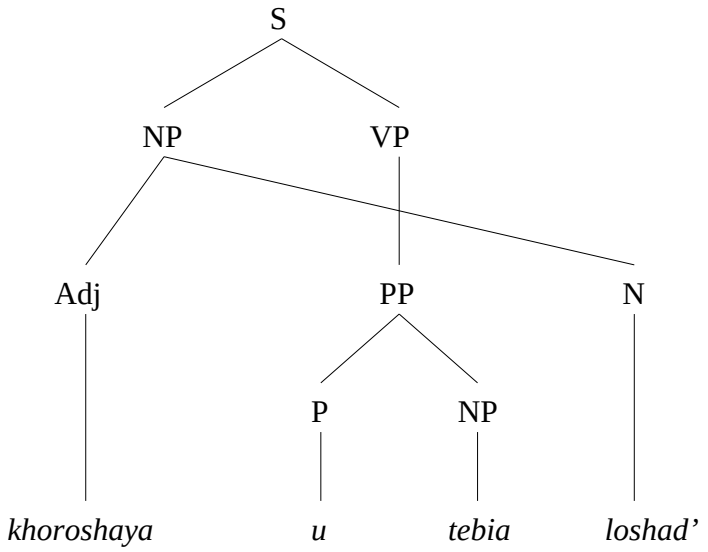
supplemented to some extent with rules of other types.

This approach has been highly successful with English, but it has trouble with variable word order for two reasons. First, the order of the constituents is variable. Second, and more seriously, variable word order often results in discontinuous constituents. Consider the following sentence, from a short story by Lermontov:

khoroshaya	*u*	*tebia*	*loshad'*.
nom.		pobj.	nom.
good	with	you	horse

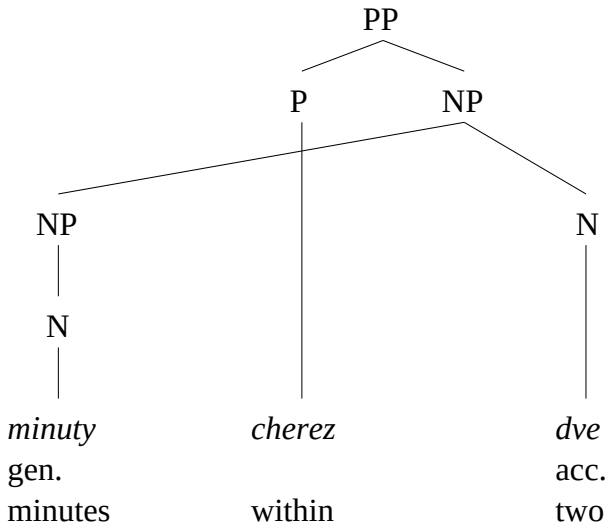
='You have a good horse.'
(lit. 'A good horse is with you.')

Here 'good' and 'horse' should, on any reasonable analysis, form a constituent. Yet 'you', which is not part of this constituent, intervenes. A phrase-structure tree for this sentence would have crossing branches, which phrase-structure grammar disallows:



(Russian omits the word *is*, allowing a VP to consist of a bare predicative NP, PP, or AdjP.)

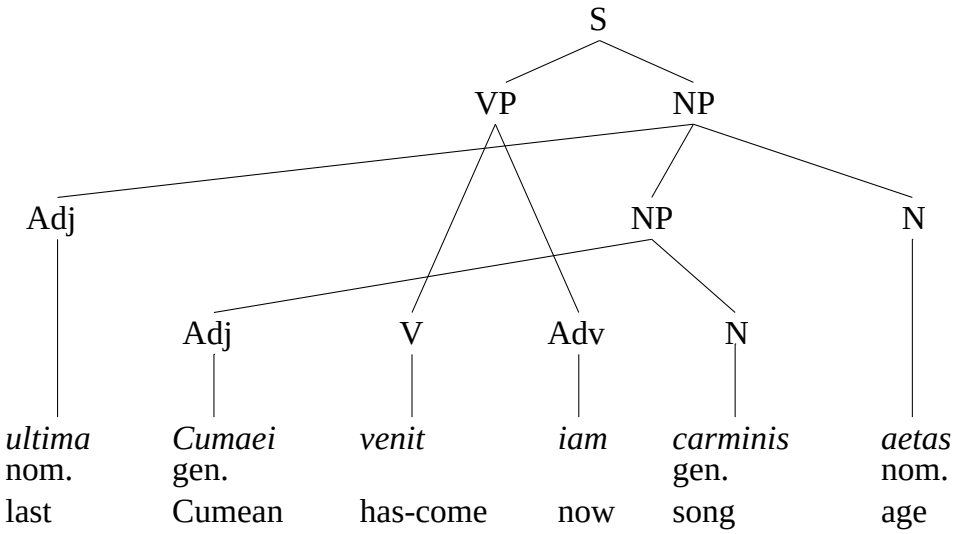
Even the object of a preposition can be broken up, as in the example



='within two minutes'
(lit. 'within a pair of minutes')

from the same short story.

In Latin poetry, extremely scrambled sentences are common. For example:



= 'The last epoch of the Cumean song has now arrived'
(Vergil, *Eclogues* IV.4)

Siewierska (1988) gives examples of discontinuity in Walbiri and other languages.

2.3 Scrambling transformations do not solve the problem

One way to handle discontinuous constituents is to write phrase-structure rules that generate them whole and then break them up with syntactic transformations or "scrambling rules." Twenty years ago, this was the preferred analysis (see e.g. Ross 1967). But such a theory claims that variable-order languages are intrinsically more complex than fixed-order languages — in fact that a variable-order language *is* a fixed-order language plus a possibly immense set of scrambling rules.

If this were so, one would expect languages burdened by word order variability to become "simpler" (i.e., more fixed) over time, but this does not necessarily happen. Though the Indo-European languages are generally drifting toward fixed order, languages in other families, such as Finnish, are developing more elaborate case systems that increase the possibilities for word order variation.

Intuitively, word order variability may well be a *simplification* of the syntax that is compensated by a more elaborate morphology. That is, a variable- word-order language is one with a lack of word-order rules, not a superabundance of them.

More importantly, a transformational analysis is worse than no help at all for parsing. Because transformations are tree-to-tree mappings, a parser can only undo a transformation if it has already recognized (parsed) the tree structure that represents the output of the transformation. That is, the only way to parse a transformed sentence is to undo the transformation—but the only way to undo the transformation is to parse the sentence first.

2.4 ID/LP formalism cannot handle discontinuous constituents

Recently, several linguists have proposed handling variable word order by splitting phrase-structure rules into two components: immediate dominance rules (ID-rules) that say what a phrase consists of, and linear precedence rules (LP-rules) that state the order in which the constituents appear.

ID/LP framework has been quite successful in capturing generalizations about word order in fixed-order languages. Gazdar et al. (1985), for instance, account for the whole of English word order with just three LP rules.

Another claim often made for ID/LP formalism is that it accounts for word order variability. If the relative order of a set of constituents is unstated by the LP rules, they are allowed to occur in any order.

But this is not enough. Removing the LP rules merely allows reordering the elements within each constituent, i.e., the nodes immediately dominated by a single node. It still does not account for discontinuous constituents. Faced with substantial word order variation, the

ID/LP grammarian must replace the transformationalist's scrambling rules with "flattening rules" that simply discard constituent structure and make nearly all words hang directly from the S node.

Flattening was pioneered and then rejected by Uszkoreit (1986a, 1987). The problem is that a "flat" structure is no structure at all — or to put it differently, a tree that shows all the words hanging from the same node is not really a tree at all; it claims only that the words form a sentence, and begs the question of what relations interconnect them.

Because of this, ID/LP parsing algorithms such as those discussed by Evans (1987) are not adequate for variable word order languages. Nor is the variable-word-order ATN parsing technique discussed by Woods (1987), which is equivalent to a form of ID/LP grammar.

2.5 Nonconfigurationality does not solve the problem

Chomsky (1981), Hale (1983), and others have split the languages of the world sharply into two types: 'configurational' languages, such as English, in which grammatical relations are defined by tree structure and word order is fixed; and 'non-configurational' languages, such as Walbiri, in which tree structure is less important and word order may or may not be variable.

Like flattening, nonconfigurationality begs the question of how to represent structure in, and how to parse, a non-configurational language. Kashket (1986) has developed a free-word-order parser consistent with Chomsky's theories, but it works rather like a dependency parser, searching through the input string for arguments of each word.

More importantly, many linguists remain unconvinced that there is a distinction between configurational and non-configurational languages. Siewierska (1988) cites extensive evidence that languages form a continuum from fully fixed to fully variable word order, with no sharp transition from one type to the other.

2.6 Current parsing technology is limited by the limitations of phrase-structure grammar

Almost all known parsing algorithms are based on constituency grammars. As a result, they have trouble handling variable word order. The normal practice among researchers in Germany and Japan is to build parsers for English — which fits the constituency model relatively well — rather than for their own languages, which do not (e.g., Kolb 1987, Matsumoto et al. 1983, Tomita 1986). A business magazine recently reported that the Japanese are developing successful computer programs to translate English into Japanese but cannot go the other way because Japanese is much harder to parse (Wood 1987).

Constituency grammars are popular for parsing largely because they are easily modeled by the context-free phrase-structure grammars (CF PSGs) of formal language theory. Thus efficient parsing algorithms are available and the complexity of parsing task is easy to study.

CF PSGs are not entirely adequate for human language, and linguists' concern has therefore been how to augment them. The major augmentations include transformations (Chomsky 1957), complex symbols (Chomsky 1965), reentrant feature structures (Kaplan and Bresnan 1982), and slash features to denote missing elements (Gazdar et al. 1985). Gazdar's theory claims as a triumph that it is closer to a CF PSG than any previous viable linguistic theory.

However, one should remember that formal language theory — the branch of mathematics from which PSG came — has nothing to do with languages, i.e., communication systems. Formal languages are called languages only metaphorically (because they have grammars). They are strings of symbols, and their most important use is to represent sequences of operations performed by a machine. It is by no means obvious that formal systems developed for this purpose should be immediately applicable to human languages.

A much more satisfactory approach is to admit that constituents are sometimes discontinuous and to develop a suitable representation and parsing strategy. Recently, discontinuous constituents have become an important issue in linguistic theory (Huck and Ojeda 1987).

The advantages of dependency parsers for dealing with discontinuity have not been generally recognized. Early computational linguists worked with dependency grammars, but only in forms that were easily convertible to PSGs and could be parsed by essentially the same techniques (Hays and Ziehe 1960, Hays 1964, Bobrow 1967, Robinson 1970). Most present-day dependency parsers impose an adjacency condition that explicitly forbids discontinuous constituents (Starosta and Nomura 1986; Fraser 1989; Hudson 1989; but not Hellwig 1986 and possibly not Jppinen et al. 1986 and Schubert 1987).

3 Dependency grammar (DG)

3.1 Dependency grammar analyzes structure as word-to-word links

The alternative to phrase structure grammar is to analyze a sentence by establishing links between individual words, specifying the type of link in each case. Thus we might say that, in "The dog sees a cat,"

dog is the subject of *sees*

cat is the object of *sees*

the modifies *dog*

a modifies *cat*

or, speaking more formally,

dog depends on *sees* as subject

cat depends on *sees* as object

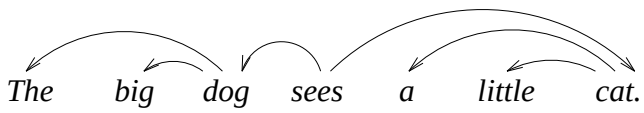
the depends on *dog* as determiner

a depends on *cat* as determiner

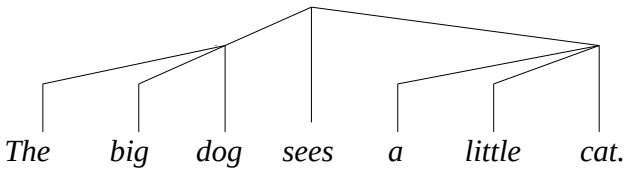
and *sees* is the head of the whole structure, since it does not depend on or modify anything else.

This is dependency grammar. It has a long and distinguished history, having been used by traditional grammarians at least since the Middle Ages (Covington 1984). The first modern treatment is that of Tesnire (1953, 1959). Present-day syntacticians who advocate dependency grammar include Baum (1976), Bauer (1979), Hudson (1980, 1984), Tarvainen (1982), Shaumyan (1987),[2] Schubert (1987), Mel'cŭk (1988), Starosta (1988), and Fraser (1989). Moreover, as I shall point out below, the latest constituency grammars include constraints that bring them closer to dependency grammar.

There are several ways to represent a dependency analysis graphically. We can annotate the sentence with arrows pointing from head to dependent:

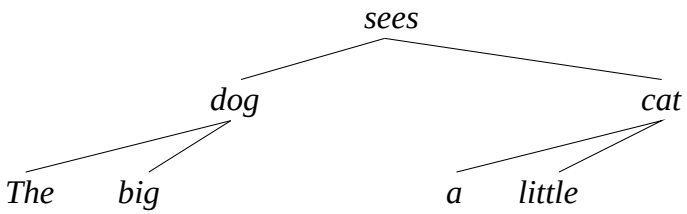


Or we can draw a "dependency tree" or D-tree, in which each head is represented by a node placed higher than that of its dependents:

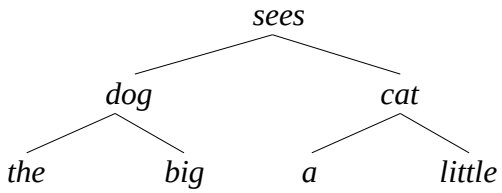


This is equivalent to:

[2]Shaumyan uses a combination of dependency and constituency representations.



Following Tesnière, we can make the tree neater by discarding the information about word order and centering each node above its dependents:



This, in turn, can be represented in outline-like form using indentation rather than lines to indicate hierarchy:

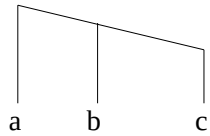
```
sees
  dog
    the
    big
  cat
    a
    little
```

This last notation is particularly convenient for computer output. It requires no line drawing, and annotations can be printed after each word.

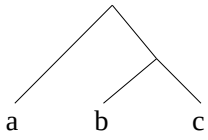
3.2 Modern linguistic theories are evolving toward dependency grammar

Every dependency analysis that specifies word order is equivalent to a constituency analysis that (1) has no labels or features on nonterminal nodes, and (2) picks out one node as the "head" of each phrase (Hays 1964, Gaifman 1965, Robinson 1970).

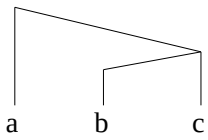
To see the equivalence, observe that the D-tree



can be converted into the constituency tree:



which has the same branching structure. To convert this back into the D-tree without loss of information, we must know whether *b* or *c* is the head of the constituent *bc*; if we incorrectly take *c* as head, we will get the incorrect D-tree:



It is obvious that *a* is the head of the larger constituent because only an individual word, not a phrase, can be a head.

Moreover, the D-tree has no nodes separate from the words themselves, whence the requirement that the corresponding constituency tree contain no features or other information on non-terminal nodes (unless of course it is copied unchanged from terminal nodes).

Most present-day syntactic theories have adopted essentially these restrictions. Jackendoff (1977) promoted the concept that every phrase has a head which is a single word. He pointed out that a PS-rule such as

noun phrase → verb + adverb

ought to be impossible even though classical transformational grammar permits it.

Intuitively, every noun phrase must contain a noun, every verb phrase must contain a verb, and every prepositional phrase must contain a preposition — or more generally, every X phrase must contain an X. Jackendoff's X-bar theory, which formalizes these intuitive constraints and defines a head for every phrase, has been accepted without controversy by subsequent workers in several different theoretical frameworks (Chomsky 1981, Bresnan 1982, Gazdar et al. 1985).[3]

Likewise, it is uncontroversial that most, if not all, of the features on phrasal nodes should be copied from their heads. (A plural noun phrase is headed by a plural noun; a singular verb phrase is headed by a singular verb; and so on.) Indeed, Gazdar et al. (1985) have a rule, the Head Feature Convention, to ensure that this is so. This is almost equivalent to not

[3] Admittedly, X-bar theory distinguishes multiple levels of phrasal structure, which dependency grammar cannot do. However, these multiple levels have become less popular in recent work (compare Jackendoff 1977 to Radford 1989).

making a distinction between the phrasal node and the head of the phrase (e.g., the noun and the noun phrase of which it is the head).

Finally, the latest syntactic theories put less and less emphasis on trees as a representation of structure. The emphasis is shifting toward the grammatical relations that link the head of a phrase to the other constituents of the phrase. Examples of this approach include lexical-functional grammar (Kaplan and Bresnan 1982), Chomsky's theory of government and binding (1981), head- driven phrase-structure grammar (Sag 1987), and categorial grammars (Bouma 1985, Uszkoreit 1986c, Flynn 1987, Steedman 1987).

Significantly, Uszkoreit argues that the constituents defined by categorial rules need not be continuous, and Bouma specifically addresses the Australian language Walbiri (Warlpiri), whose word order is extremely variable.

4 Unification-based dependency grammar

4.1 Dependency is a theory-dependent concept

The relation of head to dependent corresponds roughly to two concepts already well developed in grammatical theory:

(1) The dependent presupposes the presence of the head. That is, adjectives depend on nouns, not vice versa. Verbs are heads and their subcategorized arguments (subject, object, etc.) are their dependents.

(2) Semantically, functors are heads and arguments are dependents. That is, if the meaning of word X is incomplete without the meaning of word Y, but not vice versa, then X is the head and Y is the dependent. For example, the subject and object are dependents of the verb.

As Gazdar et al. (1985:189–192) have shown, this does not settle the issue, because in a semantics that includes partial functions, the choice of functor and argument is somewhat arbitrary.

In the past, dependency grammarians have tried to find some single observable property, such as optionalness, that distinguishes head from dependent in all constructions. My position is that this is a mistake. Dependency is a theoretical abstraction, and its identification depends on multiple criteria; questions about it should be resolved in the way that best captures syntactic and semantic generalizations.

4.2 D-rules specify possible relations

Following Miller (1985), I formalize a dependency grammar by writing "D-rules," i.e., rules that allow one word to depend on another. However, instead of using symbols like *N* and *V* for noun and verb, I use feature structures in the tradition of unification-based grammar (Shieber 1986).

Thus the relation of noun to adjective in Russian or Latin is described by the following rule, where *G, N,* and *C* are variables:

$$\begin{bmatrix} category{:}noun \\ gender{:}G \\ number{:}N \\ case{:}C \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}adj \\ gender{:}G \\ number{:}N \\ case{:}C \end{bmatrix}$$

That is: "A word with category *adj*, gender *G*, number *N*, and case *C* can depend on a word with category *noun*, gender *G*, number *N*, and case *C*."

This rule does not specify word order; in the rule, the head is always written first. First approximations to some other rules used by the parser are as follows:

Verb and subject:

$$\begin{bmatrix} category{:}verb \\ number{:}N \\ person{:}\ P \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}noun \\ number{:}N \\ person{:}\ P \end{bmatrix}$$

Verb and object:

$$\begin{bmatrix} category{:}verb \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}noun \\ case{:}acc \end{bmatrix}$$

Preposition modifying verb:

$$\begin{bmatrix} category{:}verb \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}prep \end{bmatrix}$$

Preposition and object:

$$\begin{bmatrix} category{:}prep \\ objcase{:}C \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}noun \\ case{:}C \end{bmatrix}$$

These rules of course ignore many details.

4.3 Unification builds and tests structures order-independently

This formalization relies crucially on unification (matching and/or merging) of feature structures. The power of unification comes from its ability to build complex objects through order-independent operations.

Each of the feature structures in the rules above is only partly instantiated — that is, the values of most features are unspecified. They will become instantiated through matching. Crucially, an uninstantiated feature has no value; it is not equivalent to a feature whose value is 0 or nil.

Two feature structures unify if (1) the features within them that are already instantiated have matching values, and (2) every feature that is instantiated in one structure but not in the other becomes instantiated to the same value in both structures. Thus a feature structure is built up with information acquired from many sources — some of it from the lexicon and some from every D-rule that successfully applies to it.

The value of a feature can itself be a feature structure, in which case the matching criterion applies recursively.

Unification is a process that succeeds or fails. If the grammar requires two structures to unify and they cannot be unified, the utterance is ungrammatical, i.e., it is not generated or parsed by the grammar.

The result of unifying a set of structures is the same regardless of the order in which the unifications are performed. This means that, in a unification- based grammar or computing system, many difficult questions about the order of operations simply fail to arise. The power of unification comes from its ability to make the most of whatever information is available at a particular time, filling in missing details (or even missing superstructures) when it becomes possible to do so.

For example, a transformational grammar might copy a feature from one location to another. This implies that the feature is in its original location before the copying takes place. A unification-based grammar, by contrast, will merge features in two locations (whether or not either one has been given a value) and, either beforehand or afterward, give a value to one of them. The question of which comes first, instantiation or copying, is irrelevant if not meaningless.

4.4 Representations of meaning are easy to build

The unification process can easily build semantic representations as it goes. Consider for example the D-rule

$$\begin{bmatrix} category{:}verb \\ number{:}N \\ person{:}P \\ semantics{:}X(Y,Z) \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}noun \\ number{:}N \\ person{:}P \\ semantics{:}Y \end{bmatrix}$$

This is just the subject-verb rule above with a crude semantic representation added.

Suppose this D-rule applies to the two words

$$\begin{bmatrix} form{:}vidit \\ category{:}verb \\ number{:}1 \\ person{:}3 \\ semantics{:}sees(U,V) \end{bmatrix} \longleftarrow \begin{bmatrix} form{:}sobaka \\ category{:}noun \\ number{:}1 \\ person{:}3 \\ semantics{:}dog \end{bmatrix}$$

Unification will give the following values to the variables:

```
N=1   P=3   X=sees   Y=U=dog   V=Z
```

and as a result the *semantics* feature of *vidit* will have the value *sees(dog,V)*, where V will get a value through further unification when the rule applies that introduces the object.

This is far from a complete theory of meaning, and the reader is referred to Shieber (1986) and to Gazdar et al. (1985) for further discussion of semantics in unification-based grammar. The point here is simply that semantics in a dependency grammar can use mechanisms that have already been extensively developed for unification-based phrase-structure grammar.

4.5 The meaning representation can ensure that obligatory dependents are present

Any dependency grammar must distinguish between optional and obligatory dependents. For instance, in English, the subject of the verb is obligatory but the adverb of manner is not:[4]

[4] Asterisks denote ungrammatical examples.

John came.
John came quickly.
**Came.*
**Came quickly.*

Moreover, if an argument is obligatory it is also unique: a verb may take several adverbs but it must have one and only one subject.

John invariably came quickly.
**John Bill Harry came.*

Obligatoriness of arguments is most easily handled through the mechanism that builds representations of meaning. After all, in the meaning, each verb can have only one value in each argument position; this ensures there will never be two subjects or two objects on a single verb.

To this one can add a requirement that the complete semantic representation, once built, must not contain any uninstantiated argument positions. This, in turn, will ensure that all the necessary arguments — subject, object, and the like — are present.

The list-valued *subcat* or *syncat* features of Shieber (1986) and others cannot be used to handle subcategorization in dependency grammar, because each D-rule brings in only one argument. Nor are the lists appropriate for a variable-word-order language, since the arguments are distinguished by inflectional form rather than by their proximity to the verb.

4.6 Restrictions can be imposed on word order

Even a variable-word-order language has some constraints on word order. For example, in Russian and Latin, the preposition must precede its object. In the sentence

devushka	*kladyot*	*knigu*	*na*	*gazetu*
nom.		acc.		acc.
girl	puts	book	on	newspaper

the object of *na* must be *gazetu*, not *knigu*, even though both are in the correct case. That is, the preposition can only combine with a noun that follows it.

We can handle this by annotating the preposition-object rule as "head first," i.e., the head must precede the object. Rules are likewise allowed to be annotated "head last," and such rules will be common in Japanese though I know of none in Russian or Latin.

Further, the prepositional phrase must be continuous; that is, all the direct or indirect dependents of the preposition must form a continuous string. Thus in

devushka	*kladyot*	*na*	*gazetu*	*knigu*
nom.			acc.	acc.
girl	puts	on	newspaper	book

the object of *na* must be *gazetu*, not *knigu*; the verb can be separated from its object but the preposition cannot. (Recall that in the example *minuty cherez dve* above, it was the NP, not the PP, that was discontinuous.)

The prototype parser does not handle contiguity requirements. One way of doing so might be to endow the preposition (for example) with a feature *contig* that is copied recursively to all its dependents, and then insist that the whole string of words bearing the same value of this feature be contiguous.

Hudson (1984) has shown that dependency grammars with sufficient contiguity and word order requirements can handle fixed-order languages such as English.

5 The parsing algorithm

5.1 The parser accepts words and tries to link them

Unlike other dependency parsers, this parser does not require constituents to be continuous, but merely prefers them so.

The parser maintains two lists, PrevWordList (containing all words that have been accepted from input) and HeadList (containing only words that are not dependents of other words). These are initially empty. At the end, HeadList will contain only one word, the head of the sentence.

Parsing is done by processing each word in the input string as follows:

(1) Search PrevWordList for a word on which the current word can depend. If there is one, establish the dependency; if there is more than one, use the most recent one on the first try; if there is none, add the current word to HeadList.

(2) Search HeadList for words that can depend on the current word (there can be any number), and establish dependencies for any that are found, removing them from HeadList as this is done.

This algorithm has been used successfully to parse Russian and Latin. To add the adjacency requirement, one would modify the two steps as follows:

(1) When looking for the word on which the current word depends, consider only the previous word and all words on which it directly or indirectly depends.

(2) When looking for potential dependents of the current word, consider only a contiguous series of members of HeadList beginning with the one most recently added.

With these requirements added, the algorithm would then be equivalent to that of Hudson (1989).

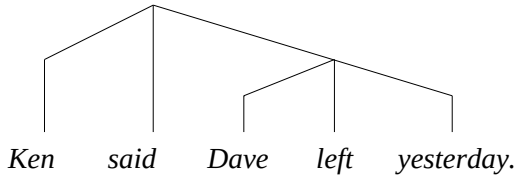
5.2 The parser prefers continuous phrases but does not require them

Comparing the two algorithms just given, it is obvious that the parser for continuous constituents is a special case of the parser for discontinuous constituents, and that, in fact, it usually tries the parses that involve continuous constituents first.

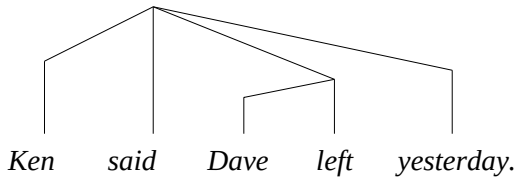
This strategy makes the parser's preferences resemble those of human beings in two respects.

First, even when discontinuous constituents are allowed, the parser has an easier time parsing continuous ones. Analogously, human languages that allow discontinuity usually do not indulge heavily in it, unless for poetic effect or some other special communicative purpose.

Second, because of its search strategy, the parser adheres to the psycholinguistic principle that near attachments are preferred (Frazier 1987). Consider Frazier's example sentence *Ken said Dave left yesterday.* Hearers prefer to interpret this as



with *yesterday* modifying *left*, rather than



with yesterday modifying said.

The parsing algorithm has the same preference. Both HeadList and PrevWordList are searched beginning with the most recently added element. Thus the parser follows the strategy

Attach each word to the nearest potential head or dependent.

This is the dependency-grammar counterpart of Frazier's principle, which says

Attach each node to the node currently under consideration if possible.

Ueda (1984) has argued on independent grounds that this principle applies to Japanese, a variable-word-order language.

5.3 Unavoidably, parsing discontinuous constituents is complex

It is well known that phrase-structure parsing of an n-word sentence with the most efficient algorithm takes, at most, time proportional to n_3. The same is true, in principle, of dependency parsing with a dependency grammar that is convertible to a phrase-structure grammar — i.e., one that does not allow discontinuity.

Parsing with discontinuity is unavoidably more complex. After all, it allows more possibilities; the parser can never be completely sure that a constituent is over or that a subsequent constituent has not yet begun. An exact analysis has not yet been carried out, but complexity of parsing with discontinuity may, in the worst case, be as high as n^n.

Three things should be emphasized. First, the extra complexity comes from allowing discontinuous constituents, not from using dependency grammar. Discontinuous constituents are necessary in some human languages, and hence unavoidable. Second, as will be shown below, worst-case complexity is irrelevant to natural language processing. Third, the complexity can be reduced by putting arbitrary limits on how far away from the current word a head or dependent can be sought. There is every reason to believe that the human brain imposes such limits on hearers' ability to understand speech, and therefore that all human languages are thus constrained.

5.4 With ambiguity and features, natural language parsing is NP-complete

Barton, Berwick, and Ristad (1987:89–96) prove that parsing is NP-complete in any phrase structure grammar that includes (1) agreement features that are copied from some nodes to others (like the agreement of subject and verb in natural language), and (2) lexical ambiguity

(the ability to rewrite more than one complex terminal symbol as the same surface form). They do this by reducing 3SAT (a well-understood theorem proving problem) to a parsing problem for such a grammar.

The dependency grammars proposed here have agreement features and lexical ambiguity. Although the details have not been worked out, it should be obvious that the same reduction can be carried out for a dependency grammar that has order and contiguity requirements. In this respect, dependency parsing is no better and no worse than phrase structure parsing.

5.5 Average-case performance is what matters

The Barton-Berwick-Ristad proof indicates that all adequate natural- language parsing algorithms have the same worst-case complexity, i.e., they are NP-complete (unless of course some of them turn out to be worse). Fortunately, worst cases in natural language are quite rare. They do exist; an example is the English sentence

BUFFALO BUFFALO BUFFALO BUFFALO BUFFALO

which has the same structure as "Boston cattle bewilder Boston cattle." Once given the structure, human beings have no difficulty interpreting the sentence and seeing that it is grammatical, though they find it extremely difficult to discover the structure without help.

The moral is that even the parsers in our heads do not perform well in the worst case. Average-case complexity is much more important. One way of limiting the average-case complexity of dependency parsing is to place a limit on, for example, the maximum size of HeadList and/or PrevWordList. This will prohibit massive inversions of word order and wide separation of related constituents — exactly the things that are rare or impermissible even in free-word-order languages.

6 The implementation

6.1 The parser is written in IBM VM/Prolog

The present implementation uses IBM VM/Programming in Logic ('VM/Prolog' for short) on the IBM 3090 Model 400-2VF at the University of Georgia. It uses the IBM 'Mixed' syntax, which closely resembles the standard Edinburgh dialect of Prolog described by Clocksin and Mellish (1981).

Prolog is usually thought of as a language for automated reasoning and expert systems (Kowalski 1979, Walker 1987). Nonetheless, it originated as a language for writing parsers

(Colmerauer 1973) and remains eminently suited for this purpose.

Three factors make Prolog ideal for natural language processing. First, Prolog is a language for data as well as operations. Every data object that can exist in the language has a written representation.[5] Complex data structures can be created gradually and straightforwardly, with no need to declare them in advance or perform system calls to allocate memory. Lisp-like lists, decomposable into head and tail, are one of the many structuring devices available.

Second, Prolog is designed to consider multiple alternative paths of computation. Parsers typically have to try many alternatives in order to parse a sentence successfully, and in Prolog this is almost automatic. A procedure can be given multiple definitions to express alternative ways of solving a problem. Every computation either succeeds or fails, and if a computation fails, execution backs up to the most recent untried alternative and proceeds forward again. The programmer can put in instructions ('cuts') to suppress backtracking where it is not wanted.

Third, a form of unification is built into Prolog. Prolog unification is not identical to feature structure unification, but the basic idea is the same: make the structures alike by instantiating (giving values to) variables. For example, the list `[a,b,X]` can unify with `[Y,b,c]` with the instantiations `X=c, Y=a`. However, `[X,b,X]` cannot unify with `[a,b,c]` because X cannot take on two values at once.

6.2 GULP extends Prolog by adding feature structure unification

Prolog unification differs from feature structure unification in one crucial way: Prolog identifies corresponding features by position, whereas in feature structures, features are identified by name.

This is a substantial obstacle for implementing unification-based grammars. The grammatical theory requires that any feature structure should be unifiable with any other unless feature values prevent it. This means every feature structure must reserve a position for every feature that occurs in the grammar, even though only one or two of them are mentioned in any specific instance. To represent

$$\begin{bmatrix} case{:}nom \\ num{:}sing \end{bmatrix}$$

the programmer has to write something like

[5] Except for certain pathological structures that contain pointers to themselves. These structures are not normally created and some Prolog implementations treat them all as runtime errors.

```
[V1,V2,V3,V4,V5,V6,nom,V8,V9,V10,V11,sing,V13,V14,V15]
```

if the grammar uses a total of 15 features. Typographical errors are inevitable.

There are two ways out of the dilemma: modify the Prolog unifier, or modify Prolog syntax. If the unifier were modified, it would be possible to write something like `[case(nom),num(sing)]` and have the unifier figure out that, for example, this is supposed to match `[num(N),case(nom),pers(3)]`. The trouble is that this approach really slows down the program; all the extra work has to be done at run time whenever a unification is attempted.

GULP (Covington 1987, 1989) is an extension of Prolog that modifies the syntax instead. The programmer writes feature structures such as:[6]

```
case % nom %% num % sing
```

and GULP preprocesses the Prolog program to convert these into list-like structures in which features are identified by position rather than by name. The value of a feature can be any Prolog object, including another feature structure. A routine is provided to display feature structures in a neatly indented style.

6.3 The implementation consists of a grammar, a lexicon, and a parser

6.3.1 Feature set

In the prototype parsers for Russian and Latin,[7] Each word is represented by a feature structure. The features used are:

phon — The phonological or orthographic form of the word.

cat — The syntactic category (noun, verb, etc.).

case, num, gen, pers — Grammatical agreement features (case, number, gender, and person). For brevity, *case* is used on the preposition to mark the case required in the object; this lacks generality, because other words (e.g., participles or the adjective 'similar') can be in one case while taking a complement in another.

[6] In the original ASCII version of GULP, `case:nom :: num:sing`. The colon is already used for another purpose in VM/Prolog and although VM/Prolog is highly modifiable, the requisite moidifications have not yet been done.

[7] Only the Russian parser runs on the 3090; the Latin parser runs (considerably lower!) on a PS/2 Model 50 using ALS Prolog.

id — An identifying number assigned to each word in the input string so that separate occurrences of the same word form will not unify with each other.

subj, obj — If the word is a verb, these become instantiated to the values of the *id* features of its subject and object respectively. These are merely stand-ins for argument positions in the semantic component that has not been implemented.

gloss — An English translation of the word, for annotating output.

gr — The grammatical relation borne by the word to its head (subject, object, modifier, etc.); for annotating output. This identifies the D-rule that was used to establish the relationship.

dep — An open list (i.e., a list with an uninstantiated tail) containing pointers to the full feature structures of all of the word's dependents. The whole dependency tree can be traced by recursively following the *dep* feature of the main verb.

6.3.2 Lexical entries

Since this parser was built purely to examine a syntactic problem, its lexical component ignores morphology and simply lists every form of every word. Part of the lexicon is shown in Listing 1 (at the end of this paper). In conventional notation, the lexical entry for Russian sobaka, for example, is:

$$\begin{bmatrix} phon{:}sobaka \\ cat{:}noun \\ gloss{:}\text{`}dog\text{'} \\ case{:}nom \\ num{:}sg \\ gen{:}fem \end{bmatrix}$$

On input, a string of words is converted by the lexical scan procedure into a list of feature structures, each of them only partly instantiated. (For instance, the *id* and *dep* features are not instantiated in the structure above.) The parser instantiates the structures further as it does its work.

6.3.3 D-rules

D-rules are stored as Prolog clauses with the principal functor '<<', which is written between two feature structures. Recall that a feature structure is a series of feature-value pairs linked by '%%' and a feature is linked to its value by '%'. Using all these notational devices, the

adjective-noun rule

$$\begin{bmatrix} category{:}noun \\ gender{:}\ G \\ number{:}N \\ case{:}C \end{bmatrix} \longleftarrow \begin{bmatrix} category{:}adj \\ gender{:}\ G \\ number{:}N \\ case{:}C \end{bmatrix}$$

would be written in Prolog as:

```
cat%noun %% gen%G %% num%N %% case%C
                  << cat%adj %% gen%G %% num%N %% case%C.
```

The complete set of D-rules is shown in Listing 2.

6.3.4 Parsing process and output

The parser implements the algorithm described in section 4.1 above. Crucially, HeadList and PrevWordList are really lists of pointers. The same word can appear in both lists, and when this happens, there is actually only one copy of the word in memory. Thus its features can be instantiated regardless of which list the word was accessed through. At the end, HeadList has only one element, the main verb.

Figure 3 shows a sample of the parser's output, which is displayed by following the *dep* features from word to word to obtain the complete dependency network. The values of the *phon, gloss*, and *gr* features are displayed for each word.

Because it is written in Prolog, the parser automatically has the ability to backtrack and try alternatives. In Figure 3, this is put to good use to find two parses for an ambiguous sentence.

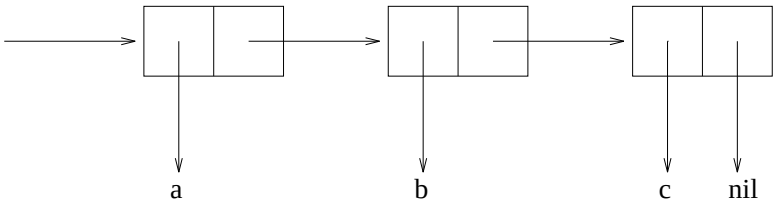
7 Evaluation of the IBM 3090 environment

7.1 Parsing (and natural language processing generally) have specific machine requirements

Parsing is one of many applications that fall into the realm of *symbolic computing* because the objects manipulated are not numbers, nor character strings, but rather abstract symbols to which the programmer assigns a meaning. The leading languages for symbolic computation

are Prolog and Lisp. Symbolic computation can be done in many other languages, but it is cumbersome.

Symbolic computation requires the ability to create complex data structures of arbitrary shape at run time — parse trees, feature structures, nested lists, and the like. Necessarily, such structures occupy noncontiguous memory locations and are held together by pointers. For example, the simplest way to represent the list `(a . (b . (c . nil)))` (more commonly written `[a,b,c]`) is the following:



This preserves the essential properties of the list: it can be recognized and processed one element at a time, without knowing the total length; it can be broken into head and tail at any point; and if the last link is uninstantiated, the list can be lengthened by instantiating it. (Prolog actually uses a more complex representation of a list with additional pointers to the dot.)

Far from wasting time, pointer references normally speed up symbolic computation. The reason is that it is faster to compare pointers to atomic symbols than to compare the symbols themselves. Accordingly, it is standard practice to *tokenize* all programs and data, i.e., replace all atoms with pointers to a symbol table.

The execution of a symbolic program consists largely of pointer dereferencing, comparisons, and conditional branches. There is very little arithmetic; the objects being processed are not numbers, and even their addresses are not calculated but rather looked up via pointers.

These requirements are not artifacts of using Prolog or Lisp; they are imposed by the applications. Small non-numeric programs, including some compilers, have been written using conventional data structures (arrays, character strings, etc.), but for larger applications, the advantages of symbolic computation are overwhelming. This is particularly the case in parsing natural language because, unlike a programming language, English cannot be designed to fit a simple transition network or require only one character of lookahead.

7.2 The IBM 3090 is well suited to symbolic computing

The IBM 3090 is a promising machine for symbolic computing because, compared to other supercomputers, it is much less narrowly specialized for non-symbolic applications. By contrast, other supercomputers are specialized for vector arithmetic (Cray-1, ETA-10) or for multiprocessing with relatively small amounts of fast memory on each CPU (Intel iPSC,

Control Data CYBERPLUS; see Stearns and Covington 1987).

By contrast, the IBM 3090 reflects the System/360 and System/370 heritage of design for all-purpose computing (Padegs 1981, Tucker 1986). Some of its "super" features are designed to speed up all types of programs, not just numeric applications. Perhaps more importantly, the 3090 imposes no penalty for the use of instructions for which it was not specifically optimized.

Symbolic computing, and especially natural language processing, requires a machine with large memory, fast access to arbitrary non-consecutive memory locations, and the ability to execute conditional branches rapidly.

The IBM 3090 meets all these requirements. At the University of Georgia, a program can run in a 100-megabyte virtual memory region if needed. The prototype parser uses only 6 megabytes and could probably get by with less, but it is only a small part of a foreseeable integrated natural language processing system. (Natural language processing is not an end in itself — it will ultimately be the user interface to some other application, which needs space of its own to run in.)

The large address space of the 3090 and the ease of access to non-contiguous locations facilitate the use of the data structures needed in symbolic computation. Non-contiguity of memory locations is no impediment to caching. This is important because data structres are allocated piece by piece and accessed by pointers. There are no arrays or vectors in Prolog, and arrays are uncommon in other symbolic processing languages.

Finally, the 3090 instruction pipeline can prefetch along both alternatives of a conditional branch instruction. This is important because parsing consists largely of decision making; prefetching speeds up execution considerably.

7.3 VM/Prolog is fast, but a much faster Prolog is possible

VM/Prolog is an interpreter, not a compiler. It was designed for versatility, not speed, and is respectably fast, but it dates from the first generation of full-featured Prolog implementations in the mid-1980s. Prolog implementations for other CPUs have advanced considerably since then. Table 1 (at the end of this paper) shows that, on an IBM PS/2, there is a factor of 20 difference in speed between interpreted and compiled Prolog. Whether a comparable speedup could be achieved on the 3090 is uncertain, but there is clearly room for improvement. The internals of the present interpreter have not been made public, but the interpreter may gain some of its speed from a tight inner loop that resides entirely in the 64K cache.

Even so, a compiler has to be faster than an interpreter. If a good compiler were available, and the speedup factor were indeed 10 to 20, then the 3090 would execute Prolog, not at 500 kLIPS (which is itself impressive), but at an unprecedented speed of 5 to 10 megaLIPS.

8 Remaining issues

8.1 Dependency is problematic in some constructions

There are constructions in which it is not clear which word is the head and which is the dependent. Prepositional phrases are an example. In a sentence like *He came after lunch,* it makes sense to treat *after* as a modifier of the verb (it says he came after something), and *lunch* as a required dependent of *after*.

But in some constructions the preposition seems to be much more tightly bound to the noun. For example, in Spanish the direct object of the verb is sometimes marked with the preposition *a* (which in other contexts means 'to'). Does such a direct object depend directly on the verb — in which case the preposition depends on the noun, rather than vice versa — or do Spanish verbs sometimes take a preposition instead of a noun as direct object?

There are other problematic constructions. Is the verb the head of the sentence? We have assumed so, but there is a time-honored traditional analysis that treats the subject rather than the verb as the head (Covington 1984). And what about relative clauses and other embedded sentences?

Fortunately, none of these problems is daunting. The question is which is the *best* analysis, not whether there is a *possible* analysis. In any case, the same questions of headship arise within X-bar theory and are the object of vigorous research efforts there (see e.g. Radford 1989).

8.2 Conjunctions are problematic for both DG and PSG

Conjunctions pose a special problem. In a sentence like *Joe and Max arrived,* the verb seems to have two subjects. Intuitively, *Joe and Max* forms a single unit that serves as the subject. But dependency grammar cannot describe this unit; dependency grammar can only connect the verb to one single word. Phrase-structure grammar seems to have the upper hand.

But Hudson (1988) has shown that PSG is not much better off than DG. Consider for example the sentence

> *John drank coffee at breakfast and tea at lunch.*

Here *and* joins *coffee at breakfast* with *tea at lunch.* Yet neither coffee at breakfast nor tea at lunch is a constituent or a grammatical unit. No reasonable constituency analysis comes out any better than dependency grammar.

From this, Hudson argues that conjunctions license doubling up of grammatical relations —

that is, because of *and,* the verb can take two objects and two prepositional phrases, instead of just one of each. Clearly, this analysis works just as well in DG as in PSG.

The alternative is to argue that, at least some of the time, conjunctions show the effect of a post-syntactic operation on the string of words — some kind of ellipsis or rearrangement not based on grammatical relations, analogous to the insertion of parenthetical remarks.

8.3 Word order variation affects emphasis and cohesion

As Karttunen and Kay (1984) have noted, word order is significant even when it is variable. The first element in the sentence is most likely to be the topic (the continuing concern of the discourse), and new information is introduced later in the sentence.

A dependency parser can easily keep track of the actual word order, or the position of various words relative to each other, by means of additional features. The semantic component of the grammar can use these features to identify topic and comment and to recognize other effects of word order.

9 Conclusions

Variable-word-order parsing is an important but neglected problem; progress on it is necessary if natural language processing is ever going to deal with a wide variety of languages other than English. The work reported here has shown that dependency parsing is a feasible approach to the handling of variable word order. The apparently high worst-case computational complexity of dependency parsing is not an objection because average-case rather than worst- case complexity is what matters; even the human brain does not process 'worst cases' successfully.

The technique presented here derives much of its power from unification- based grammar, a formalism developed to augment phrase-structure grammar but equally applicable to dependency grammar. By unifying feature structures, the grammar can build representations of syntax and meaning in a powerful, order- independent way.

Some questions remain to be answered — such as how to handle conjunctions and subordinate clauses in dependency grammar — but the work of Hudson, Starosta, and others has shown that satisfactory treatments are possible, and the question is now which analysis is best, rather than whether a satisfactory analysis can be obtained.

References

Barton, G. E.; Berwick, R. C.; and Ristad, E. S. (1987) *Computational complexity and natural language.* Cambridge, Massachusetts: MIT Press.

Bauer, L. (1979) Some thoughts on dependency grammar. *Linguistics* 17 (new series) 301–315.

Baum, R. (1976) *'Dependenzgrammatik': Tesnières Modell der Sprachbeschreibung in wissenschaftsgeschichtlicher und kritischer Sicht.* (Zeitschrift für romanische Philologie, Beiheft 151.) Tübingen: Max Niemeyer.

Bobrow, D. G. (1967) Syntactic theories in computer implementations. *Automated language processing*, ed. H. Borko, 215–251.

Bouma, G. (1985) Kategoriale grammatiek en het Warlpiri. *Glot* 8:227–256.

Bresnan, J. (1982a) Control and complementation. Bresnan 1982b:282–390.

Bresnan, J., ed. (1982b) *The mental representation of grammatical relations.* Cambridge, Massachusetts: MIT Press.

Chomsky, N. (1957) *Syntactic structures.* The Hague: Mouton.

— (1965) *Aspects of the theory of syntax.* Cambridge, Massachusetts: MIT Press.

— (1981) *Lectures on government and binding.* Dordrecht: Foris.

Clocksin, W. F., and Mellish, C. S. (1981) *Programming in Prolog.* Berlin: Springer.

Colmerauer, A. (1973) *Les Systemes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur.* Publication interne No. 43, Département d'Informatique, Université de Montréal.

Covington, M. A. (1984) *Syntactic theory in the High Middle Ages.* Cambridge University Press.

— (1987) *GULP 1.1: an extension of Prolog for unification-based grammar.* Research report 00–0021, Advanced Computational Methods Center, University of Georgia.

— (1989) *GULP 2.0: an extension of Prolog for unification-based grammar.* Research report AI–1989–01, Artificial Intelligence Programs, University of Georgia.

Covington, M. A., and Vellino, A. (1986) Prolog arrives. *PC Tech Journal* 4.11:52–69.

Evans, R. (1987) Direct interpretations of the GPSG formalism. J. Hallam and C. Mellish, eds., *Advances in artificial intelligence* (Proceedings of the 1987 AISB conference). Chichester: Wiley.

Flynn, M. (1987) Categorial grammar and the domain specificity of universal grammar. Garfield 1987:173–186.

Fraser, N. M. (1989) Parsing and dependency grammar. *UCL Working Papers in Linguistics*, vol. 1, 296–319. University College London.

Frazier, L. (1987) Theories of sentence processing. Garfield 1987:291–307.

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

Garfield, J. L., ed. (1987) *Modularity in knowledge representation and natural- langauge understanding.* Cambridge, Massachusetts: MIT Press.

Gazdar, G.; Klein, E.; Pullum, G.; and Sag, I. (1985) *Generalized phrase structure grammar.* Cambridge, Massachusetts: Harvard University Press.

Hale, K. (1983) Warlpiri and the grammar of non-configurational languages. *Natural Language and Linguistic Theory* 1:5–47.

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

— (1966) Parsing. *Readings in automatic language processing,* ed. D. G. Hays, 73–82. New York: American Elsevier.

Hays, D. G., and Ziehe, T. W. (1960) *Studies in machine translation — 10: Russian sentence-structure determination.* Project RAND Research Memorandum RM–2538. Santa Monica: RAND Corporation.

Hellwig, P. (1986) Dependency unification grammar. *Proceedings, COLING 86,* 195–198.

Huck, G. J., and Ojeda, A. E., eds. (1987) *Syntax and semantics,* vol. 20: *Discontinuous constituency.* Orlando: Academic Press.

Hudson, R. A. (1980) Constituency and dependency. *Linguistics* 18 (n.s.) 179–198.

— (1984) *Word grammar.* Oxford: Blackwell.

— (1988) Coordination and grammatical relations. *Journal of Linguistics* 24:303–342.

— (1989) Towards a computer-testable Word Grammar of English. *UCL Working Papers in Linguistics,* vol. 1, 321–339. University College London.

IBM (1985) *VM/Programming in Logic program description/operations manual.* IBM publication SH20–6541–0.

Jackendoff, R. (1977) *X′ syntax: a study of phrase structure.* Cambridge, Massachusetts: MIT Press.

Jäppinen, H.; Lehtola, A.; and Valkonen, K. (1986) Functional structures for parsing dependency constraints. *Proceedings, COLING 86,* 461–463.

Kaplan, R. M., and Bresnan, J. (1982) Lexical–functional grammar: a formal system for grammatical representation. Bresnan 1982b:173–281.

Karttunen, L., and Kay, M. (1984) Parsing in a free word order language. Dowty, D. R., et al., eds., *Natural language parsing,* 279–306. Cambridge University Press.

Kashket, M. B. (1986) Parsing a free-word-order language: Warlpiri. *Proceedings, 24th Annual Meeting of the Association for Computational Linguistics,* 60–66.

Kilbury, J. (1984) *Earley-basierte Algorithmen für direktes Parsen mit ID/LP-Grammatiken.* KIT–Report 16, Technische Universität Berlin.

Kolb, H.-P. (1987) Diskursrepräsentationstheorie und Deduktion. *Linguistische Berichte* 110:247–282.

Kowalski, R. (1979) *Logic for problem solving.* New York: North–Holland.

Matsumoto, Y.; Tanaka, H.; Hirakawa, H.; Miyoshi, H.; and Yasukawa, H. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing* 1:145–158.

Mel'cuk, I. A. (1988) *Dependency syntax: theory and practice.* Albany: State University Press of New York.

Miller, J. (1985) *Semantics and syntax.* Cambridge University Press.

Padegs, A. (1981) System/360 and beyond. *IBM Journal of Research and Development* 25:377–390.

Radford, A. (1989) *Transformational grammar.* Cambridge University Press.

Robinson, J. J. (1970) Dependency structures and transformational rules. *Language* 46:259–285.

Ross, J. R. (1967) *Constraints on variables in syntax.* Dissertation, M.I.T. Published as Infinite syntax, Norwood, N.J.: Ablex, 1986.

Sag, I. (1987) Grammatical hierarchy and linear precedence. Huck and Ojeda 1987:303–340.

Schubert, K. (1987) *Metataxis: contrastive dependency syntax for machine translation.* Dordrecht: Foris.

Shaumyan, S. (1987) *A semiotic theory of language.* Bloomington: Indiana University Press.

Shieber, S. M. (1984) Direct parsing of ID/LP grammars. *Linguistics and Philosophy* 7:135–154.

— (1986) *An introduction to unification-based approaches to grammar.* (CSLI Lecture Notes, 4.) Stanford: CSLI.

Siewierska, A. (1988) *Word order rules.* London: Croom Helm.

Starosta, S. (1988) *The case for lexicase.* London: Pinter.

Starosta, S., and Nomura, H. (1986) Lexicase parsing: a lexicon-driven approach to syntactic analysis. *Proceedings, COLING 86,* 127–132.

Stearns, R. E., and Covington, M. (1987) *Prolog on the CYBERPLUS: a feasibility study.* Research report 01–0019, Advanced Computational Methods Center, University of Georgia.

Steedman, M. (1987) Combinatory grammar and human language processing. Garfield 1987:187–210.

Tarvainen, K. (1982) *Einführung in die Dependenzgrammatik.* Tübingen: Niemeyer.

Tesnière, L. (1953) *Esquisse d'une syntaxe structurale.* Paris: Klincksieck. Cited by Robinson (1970).

— (1959) *Éléments de la syntaxe structurale.* Paris: Klincksieck.

Tomita, M. (1986) *Efficient parsing for natural language.* Boston: Kluwer.

Tucker, S. G. (1986) The IBM 3090 system: an overview. *IBM Systems Journal* 25.1:4–19.

Ueda, M. (1984) Notes on parsing in Japanese. Unpublished. Department of Linguistics, University of Massachusetts, Amherst.

Uszkoreit, H. (1986a) *Word order and constituent structure in German.* (CSLI Lecture Notes, 8.) Stanford: CSLI.

— (1986b) *Constraints on order.* Report No. CSLI–86–46. Stanford University.

— (1986c) *Categorial unification grammars.* Report No. CSLI–86–66. Stanford University.

— (1987) Linear precedence in discontinuous constituents: complex fronting in German. Huck and Ojeda 1987:405–425.

Walker, A., ed. (1987) *Knowledge systems and Prolog.* Reading, Massachusetts: Addison-Wesley.

Wood, R. C. (1987) The language advantage: Japan's machine translators rule the market. *High Technology Business* 7.11 (November), p. 17.

Woods, W. A. (1987) Grammar, augmented transition network. Shapiro, S. C., ed. *Encyclopedia of artificial intelligence* 1:323–333. New York: Wiley.

Listing 1. Part of the lexicon, which ignores morphology and simply lists every form of every word. This is a stand-in for the morphological component that would be needed in a practical system.

```
word(phon%koshka %% cat%noun %% gloss%"'cat'"  %% case%nom %% num%sg %% gen%fem).
word(phon%koshku %% cat%noun %% gloss%"'cat'"  %% case%acc %% num%sg %% gen%fem).
word(phon%koshki %% cat%noun %% gloss%"'cats'" %% case%nom %% num%pl %% gen%fem).
word(phon%koshki %% cat%noun %% gloss%"'cats'" %% case%acc %% num%pl %% gen%fem).

word(phon%sobaka %% cat%noun %% gloss%"'dog'"  %% case%nom %% num%sg %% gen%fem).
word(phon%sobaku %% cat%noun %% gloss%"'dog'"  %% case%acc %% num%sg %% gen%fem).
word(phon%sobaki %% cat%noun %% gloss%"'dogs'" %% case%nom %% num%pl %% gen%fem).
word(phon%sobaki %% cat%noun %% gloss%"'dogs'" %% case%acc %% num%pl %% gen%fem).

word(phon%vidit %% cat%verb %% gloss%"'sees'" %% num%sg %% pers%3).
word(phon%vidut %% cat%verb %% gloss%"'see'"  %% num%pl %% pers%3).

word(phon%presleduet  %% cat%verb %% gloss%"'pursues'" %% num%sg %% pers%3).
word(phon%presleduyut %% cat%verb %% gloss%"'pursue'"  %% num%pl %% pers%3).

word(phon%cherez   %% cat%prep %% gloss%"'through'" %% case%acc).
```

Listing 2. D-rules used by the prototype Russian parser.

```
cat%verb %% pers%P %% num%N %% subj%S
  << cat%noun %% case%nom %% pers%P %% num%N %% gr%subject %% id%S.

cat%verb %% obj%Ob
  << cat%noun %% case%acc %% gr%direct_object %% id%Ob.

cat%verb
  << cat%prep %% gr%modifier.

cat%noun %% case%C %% num%N %% gen%G
  <<  cat%adj %% case%C %% num%N %% gen%G %% gr%modifier.

cat%prep %% case%C %% obj%Ob %% posn%1
  <<  cat%noun %% case%C %% gr%object_of_preposition %% id%Ob.
```

Listing 3. Output of a typical parsing run. The sentence is ambiguous as to whether *belye* modifies *sobaki* or *koshki*; both parses are found.

```
[vidut,sobaki,belye,koshki,v,chornom,lesu]

Parsed structure:
vidut 'see'
  sobaki 'dogs' subject
    belye 'white' modifier
  koshki 'cats' direct_object
  v 'in' modifier
    lesu 'forest' object_of_preposition
      chornom 'black' modifier

Parsed structure:
vidut 'see'
  sobaki 'dogs' subject
  koshki 'cats' direct_object
    belye 'white' modifier
  v 'in' modifier
    lesu 'forest' object_of_preposition
      chornom 'black' modifier
```

Table 1. Comparative speed of several Prolog implementations.

	Search-and backtrack	256-element list reversal
IBM PS/2-50		
Arity Prolog 4.1	21.6 sec	47.1 sec
(interpreted)	1.5 kLIPS	0.7 kLIPS
IBM PS/2-50		
ALS Prolog 1.2	1.2 sec	2.5 sec
(compiled)	27.5 kLIPS	13.2 kLIPS
Sun Sparcstation		
Quintus Prolog	0.166 sec	0.133 sec
(compiled)	197 kLIPS	130 kLIPS
IBM 3090-400		
VM/Prolog	0.066 sec	0.297 sec
(interpreted)	496 kLIPS	112 kLIPS

Benchmark programs are from Covington and Vellino (1986).
kLIPS = thousand logical inferences per second.