A flexible graph-unification formalism and its application to natural-language processing

by Gosse Bouma
Esther König
Hans Uszkoreit

A graph-unification-based representation language is described that was developed as the grammar formalism for the LILOG research project at IBM Germany. The Stuttgart Unification Formalism (STUF) differs from its predecessors in its higher flexibility and its algebraic structure. It is well suited for the implementation of rather different linguistic approaches, but is currently employed mainly in the development of Categorial Unification Grammars with a lexicalized compositional semantics. Examples from the syntactic and semantic processing of natural language are used to illustrate the virtues of the formalism and of our lexicalist approach to linguistic analysis.

1. Introduction

The theoretical and computational virtues of unification-based grammar formalisms have been heralded in the recent literature on computational linguistics. These advantages have led many researchers to adopt this paradigm in their theoretical models as well as in their computer implementations. Linguistic frameworks that utilize the concept of unification, such as Lexical Functional Grammar and Generalized Phrase Structure Grammar, have become well-respected competitors of more traditional theories.

In this paper, we seek to demonstrate how a highly flexible graph-unification formalism can be employed for the development and implementation of a strongly lexicalist theory of syntactic and semantic processing.

The STuttgart Unification Formalism (STUF), which has been developed for IBM Germany's Project LILOG,* is not committed to any particular grammatical theory. It provides encoding strategies of syntax rules and lexical entries from several linguistic frameworks, such as Generalized Phrase Structure Grammar (GPSG) [1], Lexical Functional Grammar (LFG) [2], Augmented Phrase Structure Grammar (APSG) [3], Head-driven Phrase Structure Grammar (HPSG) [4], Categorial Grammar (CG) [5], and Categorial Unification Grammar (CUG) [6]. This is possible because syntactic constituents, lexical entries, and various rule types (such as ordinary phrase-structure rules, metarules, linear precedence rules, and lexical rules) are represented in a uniform way. The directed graph, much as in other unification grammar systems, forms the basis for the representation. Yet, STUF not only permits combining

* Project LILOG in the Division of Science and Technology of IBM Germany has as its main goal the use and development of advanced LInguistic and LOGical methods for the understanding of German through a knowledge-based program system.

solutions from different linguistic frameworks, it also provides an algebraic notation for the specification of graphs that exhibits a high degree of expressive power and semantic clarity.

In STUF, graphs can be specified in terms of other graphs, which are either called by name or constructed by description. Graphs may be called within other graphs at any level of embedding. A basic concept in the specification and processing of graphs is graph application. A certain restricted class of functions from graphs to graphs may themselves be encoded as graphs. These function or functor graphs can then be applied to argument graphs to yield value graphs. Graph application is used for Categorial Grammar rules, lexical rules, and GPSG metarules, and for the construction of the semantics.

A first test implementation in VM-PROLOG incorporated the basic notions and notations of the formalism. The implementation is currently being expanded to cover all of STUF.

In this paper, we concentrate on the application of the formalism. Although STUF is to a large extent theory-neutral, in writing actual grammars we favor a highly lexical brand of grammar that has its roots in Categorial Unification Grammar, a combination of Categorial Grammar and Unification Grammar. In this type of grammar, the only rules that combine syntactic constituents are functional application and functional composition, which are implemented in STUF as cases of graph application. This means that all combinations of constituents are binary. One of the two combining elements is the functor; the other is the argument. Traditional categorial grammars employing only functional application have often been criticized by linguists for their lack of syntactic sophistication. The combination of graph unification and functional application or composition has added the missing expressive power. The STUF formalism in turn provides a language that is better suited for CUGs than previous unification grammar models.

Our exposition starts with an outline of the basic principles and notations of the formalism in Section 2.

In Section 3, examples from English are used to demonstrate the advantages of our approach for syntax. Universal principles, constraints, and conventions that are expressed in English in most grammatical theories, such as the Head Feature Convention, can be encoded as graphs that constrain other graphs. In Section 4, examples are presented for the application of the STUF formalism to the construction of a compositional semantics. Two construction methods are discussed and compared. The first one is lambda conversion (functional application applied to lambda expressions), which parallels the use of functional application for syntactic analysis in the traditional approach of strict compositional semantics. The second one, which we call "direct construction," is a unification-based alternative to the use of lambda conversion in natural-language semantics. It is demonstrated how solutions for anaphora resolution that are adopted from Kamp's Discourse Representation Theory (DRT) [7] and Heim's File Change Semantics [8] can be integrated in our system.

Section 5, Conclusions, contains a brief discussion of the theoretical and practical relevance of the work presented.

2. Outline of the formalism

• *Graph unification and the roots of STUF*

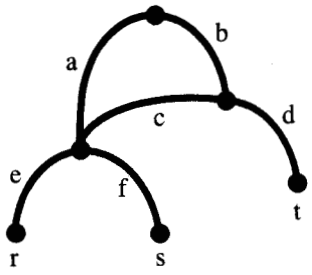
As pointed out in the Introduction, unification grammar (UG) has become one of the most successful theoretical paradigms in the area of natural-language processing. The use of graph unification in linguistic processing was introduced by Kay [9] in his Functional Unification Grammar (FUG). Although UG has its roots in computational linguistics, it is now also widely accepted in other areas of formal linguistics. The basic concepts of unification grammar have become integral parts of such widespread grammatical frameworks as LFG and GPSG. These concepts are the central notions of implemented grammar formalisms such as the PATR-II system [10] developed at SRI International and of more recent linguistic frameworks such as HPSG and CUG.

The common denominator among all the variants of UG formalisms is the use of complex attribute-value structures in the representation of linguistic units and the utilization of graph unification or some similar operation for testing, propagating, and merging of the information encoded in these structures.

The attribute-value structures can be viewed as single-rooted directed graphs with labeled edges. A graph may be empty (the empty graph), it may be an atom (an atomic graph), or it may be a set of edges whose labels are the attributes (a complex graph). Each edge points to a graph that is the value of the attribute that labels the edge. (Some formalisms also allow other types of values such as sets or strings.)

The operation of graph unification is based on the notion of subsumption. If a graph A is more general in its information content than a graph B, we say that A strictly subsumes B. B is subsumed by A if it contains at least the information contained in A. A unifies with B to a graph C if and only if C is the most general graph that is subsumed by A and B. If such a graph does not exist, i.e., if the information contained in A is not compatible with the information contained in B, the unification fails.

Many notations for such attribute-value structures have been proposed. The two most common notations are exemplified in **Figure 1** by two representations of the same structure. The index ⟨1⟩ marks a reentrancy point. The subgraph after the first occurrence of the index does not have to be repeated at the place where the index occurs again because it is already represented by the index marker. (Of



$$\begin{bmatrix} a : \langle 1 \rangle & \begin{bmatrix} e : r \\ f : s \end{bmatrix} \\ b : & \begin{bmatrix} c : \langle 1 \rangle \\ d : t \end{bmatrix} \end{bmatrix}$$

Figure 1

Two notations for directed graphs.

course, the index stands for the subgraph itself, not just for its contents.)

It is more than just the notation in which the UG formalisms differ. Even if we do not consider notation and, moreover, abstract away from the diverging encoding strategies that are introduced by the adopted linguistic theories, there is still space enough for a large variety of formalisms due to the existence of a large variety of proposed rule types, a number of operations that go beyond graph unification, and diverging approaches to phrase structure.

For all employed representations and operations, a common semantics can be constructed [11]. This means that different aspects of these formalisms can be freely combined and that only a fraction of the theoretically available possibilities have been tried out.

The existing UG formalisms fall into two categories: those that were designed for a particular grammatical framework and those that were not. The grammar formalisms of GPSG [1], HPSG [4], and LFG [2] belong to the former class. In the latter class are the powerful grammar formalisms of FUG [9] and PATR-II [10], which can be viewed as flexible computer languages for natural-language processing.

The theories, strategies, methods, and formal constructs of current formal linguistic theories are changing rapidly. Thus, a flexible research system for syntactic and semantic processing that is not dedicated to the development of a specific linguistic theory should not be based on a formalism of the first class. The philosophy behind the systems of the second class is to provide powerful tools for the representation of grammars that can accommodate a variety of strategies and analyses from more than one theory.

For our work, the use and development of advanced linguistic and logical methods for the understanding of German through a knowledge-based program system, we have adopted the latter philosophy. LILOG is a basic research project that extends into the year 1991. It is therefore of utmost importance for the goals of the project to use highly flexible systems, since this is the only chance to keep the technologies that are employed up-to-date.

The STUF system incorporates most of the basic concepts of the PATR-II formalism. (A preliminary version of the STUF formalism was presented in [12].) STUF goes beyond its predecessors in its expressive power, in the variety of permitted rule types, and in its general algebraic notation for the specification of directed graphs. The additional properties increase the flexibility of the formalism and provide a more intuitive language for representing and processing graphs.

• *Basic constructs of STUF*

Conceptually STUF consists of two components that can be modified independently of each other: the graph-specification component and the grammar-definition component. The graph-specification component provides a notation for the description of graphs together with the corresponding semantics. In our project, it is used not only in the linguistic analysis but also for the representation and management of knowledge. The grammar-definition component contains the notation for lexical entries and different types of grammar rules together with their meaning in the definition of a language.

Since all lexical entries and all types of rules are represented as graphs, a simple operation of graph declaration suffices for defining (or declaring) the appropriate graphs through the graph-specification language.

Graph declaration

Since all rules, lexical entries, and moreover all constituents in the analysis are represented as directed graphs, the notation for describing and defining such graphs plays a central role in the STUF formalism. A graph declaration consists of the graph name and of the specification of the graph separated by a colon:

graph_name : graph_specification.

The graph specification is terminated by a period. Just as in LISP and several other modern programming languages, the declaration of a variable and its value assignment are combined in one operation. However, in this language, value

assignment is in the spirit of the unification idea: The new value of the graph name is unified with its old value. If the name has no previous value, the empty graph is assumed and the new value is therefore fully determined by the graph specification. Usually, one does not want to declare a graph twice in the same grammar. Thus one might question the significance of the decision to use unification instead of overwriting in the assignment of values. But STUF is a truly declarative language. The linear order of declarations does not influence the interpretation of the grammar. STUF differs from PATR-II in this respect, since there, a second declaration of the same variable leads to a compile-time error. However, we point out to the worried grammar writer that our decision does not imply that inadvertent reuse of the same graph name must remain undetected. A good compiler for STUF should offer the possibility of notifying the user when a graph name is declared more than once.

A similar strategy should be used for unification failures during the compilation of grammars. Whereas such failures lead to compile-time errors in PATR-II, in STUF the value FAIL is assigned. (FAIL corresponds to the designated value TOP in [13].) Graphs that contain FAIL do not make sense in the kind of grammars we are working with. Here again, the compiler should notify the user of the potential source for run-time errors. Graphs that contain the graph FAIL as a subgraph have to be marked as being partially inconsistent in the implementation because they cannot be used anymore in most components of the analysis. Using the FAIL value instead of simple unification failure has a couple of advantages, which are described in [14].

Graph specification

A graph can be specified by a name or by a description. A graph name pulls in a copy of the value that has been assigned to that name. A graph description depends on the type of the graph. If the graph is empty, the graph is denoted as the empty string or as the symbol ∅. If the graph is the failure graph, it is denoted by the designated atom FAIL; if it is an atomic graph, it is denoted by the atom. The interesting part of the notation is the way in which complex graphs are described. A complete graph is seen as the result of an operation on graphs:

complex_graph :: = "(" graph_operation ")".

The most common graph operation is unification. It is denoted by a list of operands that are separated by ⊔ or by an empty operator symbol. Another graph operation is disjunction. It is written as a list of disjunct graphs separated by the disjunction operator |. The graphs that are connected by disjunction are possible values for the (sub)graph. If a nondisjunctive graph is unified with a disjunction, it is unified with every disjunct. If two disjunctive graphs are unified, the result is the cross product, i.e., the disjunction of their crosswise unifications [15].

The following graph declaration is a fictitious example for the interaction of unification and disjunction:

PastParticiple : (Verb Nonfinite (Perfective | Passive)).

The graph PastParticiple is defined as the unification of three other graphs: Verb, Nonfinite, and the disjunction of the graphs Perfective and Passive.

We have not yet provided any means for constructing complex graphs as sets of attributes and values. To this end, we need a way to introduce edges and subgraphs. The only subgraph specification that we will describe here is the path specification, which we have adopted from PATR-II. A path is written as a sequence of edge labels (attributes) enclosed by angle brackets. A path is a sequence of edges that leads to some subgraph. A simple path specification is the list of atoms that label the edges of the path. The leftmost label names an edge that starts at the root of the graph; the rightmost one belongs to the edge that immediately points to the subgraph to be specified. Thus the empty path ⟨⟩ specifies the root. Any path of length 1 specifies an immediate subgraph of the root. Examples that illustrate the path notation can be drawn from Figure 1. In the graph, there is a single path leading to the atomic graph t, which is labeled ⟨b d⟩. Two paths lead to the subgraph that is indexed with ⟨1⟩: ⟨a⟩ and ⟨b c⟩. The subgraph could be referred to by either path specification.

The formalism actually provides more means for specifying paths than just concatenation of edge labels. An important feature is the use of regular expressions in path specification that was introduced in LFG under the term "functional uncertainty." Such path specifications can denote potentially infinite sets of paths. They are especially useful for the declarative specification of sets of subgraphs. The path specification $\langle x^+ \rangle$, for example, could be used to access any subgraph that can be reached from the root by just traversing paths with the label x. As in the common notation for regular expressions, the superscripted plus sign stands for the positive Kleene-closure. The Kleene-star is permitted as well. It is also possible to define path variables, names that stand for sets of paths.

The appropriate operation for assigning values to subgraphs is the graph equation, which resembles the equation notations of LFG and PATR-II. STUF equations are sequences of graph and subgraph specifications separated by equal signs. Delimiter-omission rules specify the conditions under which equations may be written without enclosing parentheses. We do not state these rules here, but omit parentheses wherever the scope of the operator is clearly recognizable.

The semantics is rather straightforward: The operation returns a graph in which the subgraphs that are denoted by subgraph specifications are unified with each other and with the graphs denoted by the graph specifications. In contrast to the equation in PATR-II, the operation is commutative.

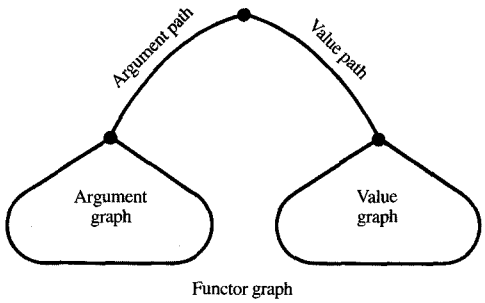


Figure 2

Schema of graph application.

The following two equations have the same meaning:

⟨agr number⟩ = pl,
pl = ⟨agr number⟩.

Both return the following graph:

[agr : [number : pl]].

The equation below merely unifies two subgraphs:

⟨number⟩ = ⟨subject number⟩.

This is the result:

$$\begin{bmatrix} \text{number} : \langle 1 \rangle \\ \text{subject} : [\text{number} : \langle 1 \rangle] \end{bmatrix}.$$

The next equation contains an embedded equation:

(⟨case⟩ = nom) = ⟨subject⟩.

This is the corresponding graph:

[subject : [case : nom]].

The graph in Figure 1 could be represented by the two following equations:

⟨a⟩ = ⟨b c⟩ = ((⟨e⟩ = r)(⟨f⟩ = s)),
⟨b d⟩ = t.

The notation does not rule out equations without any subgraph specifications. The graph specifications in such equations would be unified with each other but not with a subgraph of the graph that is returned by the equation. Such an equation would return the empty graph. Since there is an easier way to label the empty graph—actually one that does not require any writing at all—it would be a rather superfluous use of the notation. However, it is not the task of the formalism to prevent its abuse. A notation that is syntactically and semantically clean is to be preferred over one that sacrifices such features in exchange for more user guidance. Again, the structure of the implementation needs to be separated from the design of the formalism: The compiler (or interpreter) can offer guidance for the user through warnings or other notifications.

Graph application

The three operations on graphs that we have introduced so far suffice to describe the representations that are needed in the grammar. However, STUF provides another operation in addition to unification, disjunction, and graph equation, which is called graph application. Uszkoreit [6] introduces a method of representing certain functions on graphs as graphs themselves. There, this method is employed for the encoding of categorial grammars in a graph-unification formalism. (Further developments of the resulting CUGs and their efficient encoding in STUF are discussed in Section 3.) The concept of graph application is based on the insight that every complex graph can actually be viewed as encoding a set of functions from graphs to other graphs.

Graph application can be viewed as a noncommutative binary operation on graphs. The operation can be further parametrized by adding two path specifications. Thus, the operation becomes a function of four arguments—a functor graph, an argument path, an argument graph, and a value path:

ga(functor_graph, argument_path, argument_graph,
value_path).

As depicted in **Figure 2**, its value is determined as follows. Take a functor graph and unify the argument graph with the subgraph of the functor graph that is designated by the argument path. The value of the application is the subgraph of the functor graph that is designated by the value path.

Graph application is a useful way of defining graphs as being constrained by other graphs without having to include the constraining graph fully in the constrained graph. Usually, at least some information is carried over from the argument graph to the value graph.

A simple encoding of functional application uses functor graphs that contain an edge labeled "arg" leading to the argument subgraph and another one with the label "val" that leads to the value subgraph. The corresponding call of graph application is accordingly parametrized:

value_graph =
ga(functor_graph, ⟨arg⟩, argument_graph, ⟨val⟩).

The lexical rules of PATR-II can also be viewed as instances of graph application. A lexical rule contains an attribute "in" and an attribute "out." A lexical entry is unified with the value of "in." The lexical rule yields as a result the value of "out." Parts of the information contained in the original entry may be included in the new entry:

output_entry = ga(lexical_rule, ⟨in⟩, input_entry, ⟨out⟩).

A highly simplified version of the passive rule, as given in **Figure 3**, might serve as an example: The expressive power of graph application comprises several other operations on graphs that have been discussed in the literature. It is easy to see how simple binary graph unification could be expressed as an instance of graph application:

result_graph = ga(graph_1, ⟨⟩, graph_2, ⟨⟩).

Graph_2 is unified with the root of graph_1. The result is again the root. Unification with a subgraph can be expressed in a similar way:

result_graph =
ga(graph_1, subgraph_specification, graph_2, ⟨⟩).

Graph extraction yields a subgraph of another graph:

extracted_graph =
ga(main_graph, ⟨⟩, ∅, subgraph_specification).

Only a few of the potential uses of graph application will occur very frequently in the definitions of grammars. The most frequent use in our own work has been the encoding of functional application in syntax and semantics. Examples are discussed in the remainder of this paper. Because functional application occurs rather frequently, an abbreviatory convention can be introduced in the notation. If one agrees in advance on subgraph specifications for argument and value subgraphs, functional application may be simply written as functor[argument]. If, e.g., the argument subgraph is always under ⟨function argument⟩, and the value subgraph under ⟨function value⟩, then the following equivalence holds:

ga(G_f, ⟨function argument⟩, G_a, ⟨function value⟩) =
G_f[G_a].

We now leave the graph-specification component of the formalism and turn to the grammar-definition component. Other features, applications, and problems of the graph-specification language are discussed in [14].

- *Grammar component*

Lexical entries

The lexicon definition contains pairs of terminal strings and associated graphs. A lexical entry is simply a graph declaration. Here is a simplified example of a lexical entry for the English verb *like*:

like : (Verb Transitive).

Rules with omission of parentheses permit an even simpler notation:

like : Verb Transitive.

"Verb" and "Transitive" are names of graphs (templates)

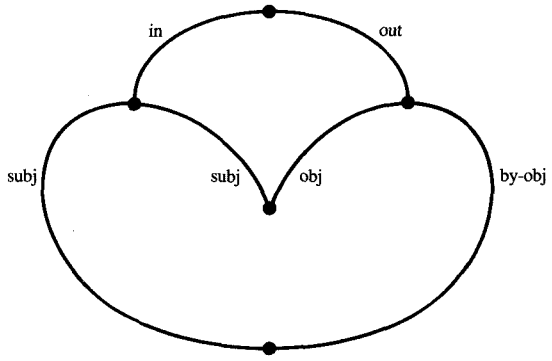


Figure 3

Graph for a simplified version of the passive rule.

whose unification yields the content of the lexical representation. (In an actual lexicon one also wants to specify the semantic content of the word and include the lexical string in the representation.)

The graphs "Verb" and "Transitive" are defined by graph declaration. "Transitive" might be declared in terms of the graphs "Subject" and "Object." The resulting modular structure of the lexicon has practical advantages that cannot be overestimated. Changes in the syntax or in the compositional semantics do not necessitate excessive lexicon editing, no matter how lexicalized the linguistic framework might be. The editing is restricted to the relevant graph definition. The same lexicon can be used for experimenting with different syntactic approaches, for it is just the definitions of certain templates that need to be exchanged.

Grammar rules

A full description of the types of grammar rules that are supported by the formalism would go far beyond the scope of this paper. Instead we will try to convey the basic principles that underlie STUF rules.

A syntax rule in STUF is a graph. Therefore, it needs to be declared as such. The simple sentence rule that is usually written as S → NP VP plus annotations might be declared as follows:

S-formation : ⟨mother cat⟩ = s
⟨daughter_1 cat⟩ = np
⟨daughter_2 cat⟩ = vp
⟨daughter_1 agr⟩ = ⟨daughter_2 agr⟩.

However, linguists are used to their traditional notation. Just as in PATR-II, we therefore permit the notation

functional application (rightward) : A → A/B B
functional application (leftward) : A → B A\B

Figure 4

Basic reduction rules.

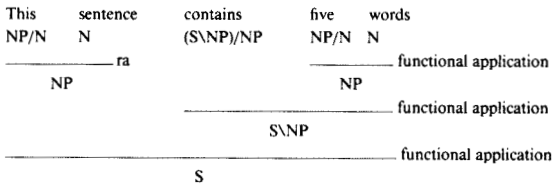


Figure 5

Sample derivation.

STUF

a. ⟨syn⟩ = n

b. ⟨syn val syn⟩ = np
⟨syn dir⟩ = right
⟨syn arg syn⟩ = n

Graph notation

a. [syn : n]

b. [syn : [val : [syn : np]
dir : right
arg : [syn : n]]]

Figure 6

Categories in STUF: (a) represents a noun, (b) a determiner.

S-formation : mother → daughter_1 daughter_2
⟨mother cat⟩ = s
⟨daughter_1 cat⟩ = np
⟨daughter_2 cat⟩ = vp
⟨daughter_1 agr⟩ = ⟨daughter_2 agr⟩.

In our example, it is obvious that the phrase-structure rule at the top has not added any information. Yet, this is only true because the information that states the immediate dominance and linear precedence relation among the three constituents that are mentioned in the rule has been omitted from the graph. Since the encoding of this information is irrelevant in the context of the current discussion, we do not demonstrate it in this article. The only aspect of the actual encoding that is of interest here is the strategy of using the phrase-structure notation as an abbreviation for a graph that adds the necessary information about the position of the constituents with respect to one another in the syntactic tree.

3. A CUG syntax in STUF

In this section we discuss how Categorial Grammar can be reformulated as a unification grammar, compatible with the STUF format. The resulting *Categorial Unification Grammar* (CUG) combines the advantages of Categorial Grammar with those of using Unification Grammar. Here we are concerned with syntax only; the relationship to semantics and parsing is discussed in Section 4.

Basic Categorial Grammar

After being invented by Ajdukiewicz [5], Categorial Grammar was, for almost forty years, studied only by logicians and mathematicians (such as Bar-Hillel, Lambek, Geach, and Montague). Linguistic interest in these grammars arose at the beginning of this decade. That Categorial Grammar is an interesting and successful framework for doing linguistics is illustrated by work on unbounded dependencies [16], X′-grammar [17], coordination [18, 19], V-raising [20], and morphology [21], to name but a few. More recent developments have led to a combination of Categorial Grammar with unification-based formalisms [6, 22, 23].

The central idea behind Categorial Grammar is the insight that not only can one analyze the semantics of natural language as consisting of functor–argument structures, but one can do this for syntax as well. A determiner, for instance, is a functor which takes a noun as argument and produces an NP as value (this is written NP/N). An intransitive verb is a functor taking an NP as argument to produce a sentence as value (written as S\NP; the backslash indicates that this functor follows rather than precedes its argument). A transitive verb takes an NP as argument to produce a phrase of the same category as intransitive verbs [(S\NP)/NP]. There are (minimally) two reduction rules governing the process which combines functors and arguments (see **Figure 4**).

A sample derivation in Categorial Grammar is presented in **Figure 5.**

Two advantages of using Categorial Grammar deserve to be mentioned. First of all, there is a clear relation between syntax and semantics: Whenever application takes place in

a. A → A/B B
b. A → B A\B
c. Rightward-Application : value → functor argument
⟨functor syn val⟩ = ⟨value⟩
⟨functor syn dir⟩ = right
⟨functor syn arg⟩ = ⟨argument⟩

(graph: [value : ⟨1⟩, functor : [syn : [val : ⟨1⟩, dir : right, arg : ⟨2⟩]], argument : ⟨2⟩])

d. Leftward-Application : value → argument functor
⟨functor syn val⟩ = ⟨value⟩
⟨functor syn dir⟩ = left
⟨functor syn arg⟩ = ⟨argument⟩

Figure 7

Reduction rules in STUF.

this: [syn : [val : [syn : np]], dir : right, arg : [syn : n]]
sentence: [syn : n]
application → [syn : np]

Figure 8

Sample derivation.

functional composition (rightward) : A/C → A/B B/C
functional composition (leftward) : A\C → B\C A\B
raising (rightward) : B/(B\A) → A
raising (leftward) : B\(B/A) → A

Figure 9

Additional rules.

syntax, functional application [usually expressed by means of lambda conversion (see the next section)] takes place in semantics as well. Second, syntactic rules are presented in a highly general form. That is, traditional phrase-structure rules are dispensed with, in favor of the rules of Figure 4. The information normally stored in phrase-structure rules and subcategorization frames is now encoded in the syntactic categories themselves.

The latter aspect of Categorial Grammar situates it firmly in the trend of increasing lexicalization in syntax [4, 9, 22]. The combination with a unification-based formalism like STUF is especially attractive here, since this formalism offers the possibility of eliminating redundancy from the lexicon by means of defining templates.

• *Categorial Unification Grammar*

Reformulating Categorial Grammar as a STUF-grammar means that the rules and categories of the grammar will have to be expressed in attribute-value structures.

Categories are either basic (for instance, S, N, and NP can be considered basic), or complex. In the latter case, they have the form A/B or A\B, where A and B must also be categories. A complex category is made up of three parts, a VALue (A), a DIRection (/ or \) and an ARGument (B). Since categories represent the syntactic properties of a phrase, we will assume that they consist of an attribute SYN, dominating either an atomic value or a complex structure (see **Figure 6**).

Rules are defined by means of a context-free skeleton and an attribute-value structure. To this end, the reduction rules in Figure 4 are represented as (highly general) PS rules **[Figure 7(a), (b)]**. In STUF, they are represented as **Figure 7(c), (d)**, respectively. The context-free portion of parts (c) and (d) serves to steer the parsing process. It expresses the fact that reduction of two categories can take place only if they dominate two adjacent elements, one of which has to be considered as functor, the other as argument. Additional information is provided by the corresponding attribute-value structure. By convention, the symbols appearing in the PS-rule match the top-level attributes in the attribute-value structure.

Combination of a determiner and a noun can now be represented as shown in **Figure 8**.

• *Extending Categorial Grammar*

Basic Categorial Grammar, using only left- and right-application, is context-free. For an adequate analysis of natural language, some additional expressive power is needed. One way to obtain this is to add rules like those of **Figure 9**, which allow for "flexible phrase structures."

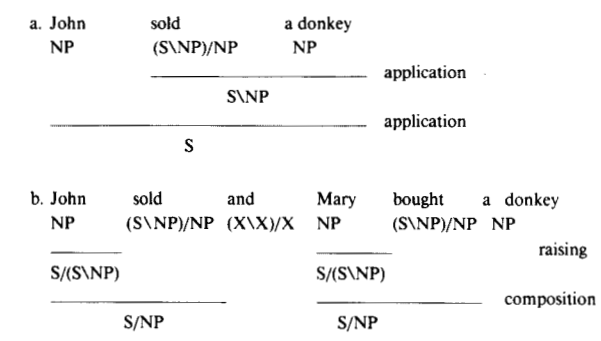


Figure 10

Coordination with raising and composition.

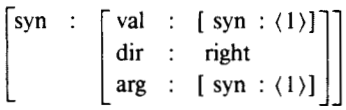


Figure 11

A partially specified category.

One kind of motivation for these rules (see [16] for more general points) is coordination, as shown in **Figure 10**.

Conventionally, a subject and a transitive verb do not form a constituent. Figure 10(b) suggests, however, that they *should* form a constituent. A Categorial Grammar with type-raising and composition is able to derive both structures.

Another way to obtain additional expressive power is to use unification. The consequences of this extension have not yet been studied extensively, but it seems, for instance, that the fact that unification naturally allows for categories such as X/X (see **Figure 11**) or (X\X)/X (as used in the example given earlier for coordination) might make composition and raising to some extent superfluous (see [6] or [24]).

From the existing literature on Unification Grammar, it is clear that there are important advantages in using unification in the morphosyntactic realm. We now show that using Categorial Grammar leads to interesting generalizations in this area as well.

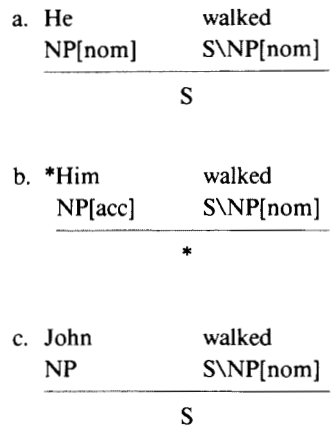


Figure 12

Case assignment.

• *Distribution of morphological information*

It is obvious that the basic system sketched earlier will have to be extended in several respects before it can be considered a full-fledged grammar. Here we show how a feature-distribution mechanism can be incorporated which is able to account for phenomena such as agreement and case assignment. (A more detailed discussion is presented in [24].) The underlying ideas for this system can be found in Bach [25, 26].

Case assignment is a process in which a functor (verb or preposition) forces its argument to be of a particular form. This can be expressed if features are added to the argument category of the verb, and to the NPs functioning as complements. **Figure 12** gives a simple example. Using unification means that a derivation will be blocked only if conflicting information is present [Figure 12(b)]. If case is left unspecified, no conflict arises [Figure 12(c)].

As shown in **Figure 13**, a similar story can be told for agreement (the only difference being that in this case it is the argument which determines the form of the functor). The addition of morphosyntactic features to the grammar is unproblematic. Categories will no longer only consist of a SYN-attribute, but will also have a MOR-attribute, dominating such attributes as CASE, AGR, and GENDER, which in turn will dominate atomic values. The category of *he* can for instance be represented as shown in **Figure 14**.

Apart from this feature-unification mechanism, we also need a system for feature percolation. For instance, in **Figure 15** the information that the complement-VP contains a bare infinitive (we need this information, since *will* subcategorizes for this property) is provided by the verb in this VP. (We have adopted the convention that features of an argument

a. He — walks
NP[3sg,nom] — S\NP[3sg,nom]
S

b. *They — walks
NP[plu,nom] — S\NP[3sg,nom]
*

Figure 13

Agreement.

$$\begin{bmatrix} \text{syn} : \text{np} \\ \text{mor} : \begin{bmatrix} \text{case} : \text{nom} \\ \text{agr} : \text{3sg} \\ \text{gender} : \text{male} \end{bmatrix} \end{bmatrix}$$

Figure 14

Representation of *he*.

are written next to the category it belongs to, whereas features ranging over complete categories are written under it.)

Thus, the feature specification [VFORM: bare] will have to be percolated from the verb *solve* to the VP. The most common assumption in this case is to assume that features will always percolate from the head of a phrase to the mother of that phrase (this is incorporated explicitly in GPSG [1] as the "Head Feature Convention," or HFC). In Categorial Grammar, a somewhat different picture arises. Usually, it will be the case that the head of the phrase is the functor within that phrase (see [27]). This means that for purposes of feature distribution one can add to the reduction-rules the path equation shown in **Figure 16**. This rule will account for the percolation in Figure 15. Cases where the head of a phrase is the argument, rather than the functor, arise if modifiers and specifiers are combined with a head. This is typically the case if a determiner combines with a noun (see **Figure 17**).

For such cases, it is assumed that specifiers and modifiers are characterized by the fact that they contain the path equation

Peter — will — solve — the problem.
NP — (NP\S)/VP[bare] — VP/NP [bare] — NP
VP [bare]
NP\S
S

Figure 15

Feature percolation.

$$\langle \text{functor mor} \rangle = \langle \text{value mor} \rangle$$

Figure 16

The functor feature convention.

the — boy
NP/N⟨1⟩ — N
⟨1⟩ — [sg,male]
NP
[sg,male]

Figure 17

Sample derivation.

$\langle \text{mor} \rangle = \langle \text{syn arg mor} \rangle$.

In Figure 17 this is made explicit by coindexing these two positions. The above path equation and the Functor Feature Convention together guarantee that the HFC will hold in CUG as well.

⟨sem lambda⟩ = ⟨sem formula arg⟩
⟨sem formula pred⟩ = sleeps

Figure 18

Simple lambda expression in STUF.

$$\begin{bmatrix} \text{syn} : \begin{bmatrix} \text{val} : & [\text{syn}: \ \text{s}\ ] \\ \text{dir} : & \text{left} \\ \text{arg} : & [\text{syn}: \ \text{np}] \end{bmatrix} \\ \text{sem} : \begin{bmatrix} \text{lambda} : \langle 1 \rangle \\ \text{formula} : \begin{bmatrix} \text{pred} : & \text{sleeps} \\ \text{arg} : & \langle 1 \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 19

Syntax and semantics of a one-place verb.

4. Constructing semantic representations in a CUG

Construction of semantic representations for natural-language sentences should be done in a systematic way. Since it is performed after the syntactic analysis of a sentence—or in parallel—the most obvious approach is to take the (output of the) parsing process as a guideline for the construction of the semantic representation of a sentence. This corresponds to the basic idea of compositional semantics. Approaches to translation and compiling techniques in computer science [28] and linguistic proposals resemble each other very much. Bach [29] states the rule-to-rule hypothesis which says that each production rule in a grammar has associated rules for semantic construction. In compiler construction theory this would be called "syntax-directed translation." Each grammar symbol has attributes which transport the information needed for constructing the translation of the input code. The set of attributes for each grammar symbol is partitioned into two disjoint sets: synthesized and inherited attributes. The values of synthesized attributes of a grammar symbol depend only on the values of its daughter nodes; inherited attributes inherit their values from attributes of mother or sister nodes in the parse tree. The single use of inherited attributes eases top-down parsing and the parallel construction of the output structure; the use of synthesized attributes in grammar eases bottom-up syntactic analysis and semantic construction.

Parsing in categorial grammars is inherently bottom-up, because all syntactic information is stored in the lexicon, and thus, there are usually no top-down expectations which the parser could exploit, apart from the very general rules of functional application (which indeed could be considered as a skeleton of production rules of context-free grammars in Chomsky Normal Form). Using a unification formalism allows for encoding attributes which are inherited among sisters. The relations among attributes of the functor and of the argument can be stated in the functional application rules. Furthermore, unification blurs the distinction between inherited and synthesized attributes, because all attribute values are expressed as (shared) pointers to graphs. In a strictly declarative grammar it is not important *when* one of these graphs pointed at is instantiated (i.e., is assigned a concrete value).

Obviously, there is both inherited and synthesized information to be dealt with in syntactic analysis. Morphological features, syntactic categories, and (partial) semantic representations can be associated with lexemes, which means that they represent synthesized information. Contextual information, e.g., representation of antecedents for anaphora resolution, could even be inherited from previous sentences when a whole text is being parsed.

• *Construction rules*

Both syntax-directed translation and compositional construction of semantic representations require a mapping of syntactic rules into construction rules. In the case of categorial grammar, there must be a semantic construction rule which corresponds to the syntactic rules characterized by functional application. Following the tradition of categorial grammars, the rule for constructing the SEMantic representation of the VALUE of a functional application is lambda conversion. For a certain subset of the traditional lambda calculus, lambda conversion can easily be defined in STUF as graph application, which we call "lambda conversion on graphs" [30].

Suppose that every category graph has a SEMantic attribute in addition to the SYNtax attribute. The value of the SEM attribute can then be a lambda expression coded in STUF (see **Figure 18**).

In order to yield the semantic representation "sleeps(pedro)" for the sentence *Pedro sleeps.* (cf. **Figure 19**), the lambda expression [lambda X sleeps(X)] has to be applied to the proper-noun translation "pedro." Exceeding the means of STUF, an additional operation "lambda conversion" could be imagined (see **Figure 20**).

Without spoiling the clarity of the STUF formalism by introducing additional arbitrary functions, lambda conversion on graphs can be encoded "statically" as in **Figure 21**.

The idea of a graph-unification formalism is to take advantage of the virtues of structure sharing in an optimal way. Structure sharing means that a subgraph can be identified in the supergraph by any representative of the class of paths leading to it. Lambda conversion on graphs accesses the LAMBDA argument graph of the FUNCTOR by the path ⟨sem lambda⟩.

Because ⟨sem formula arg⟩ = ⟨sem lambda⟩ holds by definition of the lambda formula, the location of the argument graph is also determined by the path ⟨sem formula arg⟩. By using this latter path for argument access, the category definition of one-place verbs can be reformulated, as shown in **Figure 22**, to include the rule for "direct construction" of the output representation.

The ⟨lambda⟩ path is now redundant. In this simple case, the construction rule consists only of the two path equations of **Figure 23**.

Lambda conversion on graphs is a general rule which is used for semantic construction along with functional application rules in syntax. For "direct construction" of semantic representations, there must be at least one construction rule associated with each functor category.

"Direct construction rules" are another step to lexicalization of grammatical information, as they appear in the graphs associated with lexemes.

The advantage of direct construction rules is that they allow for stating transparently the functor–argument relation for the construction of the semantic representation of the corresponding syntactic constituent. For lambda formulae the inversion of syntactic functor–argument relations for semantic purposes often must be accomplished by complicated type-raising mechanisms.

• *Treatment of contextual information*

Partial semantic representations are defined in the lexicon; therefore, they represent information transported by synthesized attributes. But (as exemplified in **Figure 24**) what happens to contextual information which has to be inherited?

In a bottom-up parse with functional application only, there is no antecedent available when the anaphoric pronoun *his* is encountered.

A syntactic solution can be found which allows for strict left-to-right parsing and therefore left-to-right propagation of contextual information. Pareschi and Steedman [31] propose using type-raising and functional composition (see also Figure 9) as additional "syntax rules." Our example would then appear as shown in **Figure 25** (with type-raising for noun phrases). In the case of forward anaphora such as the pronoun *his* in the sentence *Pedro beats his donkey*, the

Rightward-Application: value → functor argument

⟨value sem⟩ =
lambda-conversion (⟨functor sem⟩, ⟨argument sem⟩)

Figure 20

Possible encoding of lambda conversion.

Rightward-Application: value → functor argument

⟨functor sem lambda⟩ = ⟨argument sem⟩
⟨value sem⟩ = ⟨functor sem formula⟩

Figure 21

Lambda conversion on graphs.

$$\begin{bmatrix} \text{syn:} & \begin{bmatrix} \text{val} : & \begin{bmatrix} \text{syn:} & \text{s} \\ \text{sem:} & \langle 2 \rangle \end{bmatrix} \\ \text{dir} : & \text{left} \\ \text{arg} : & \begin{bmatrix} \text{syn:} & \text{np} \\ \text{sem:} & \langle 1 \rangle \end{bmatrix} \end{bmatrix} \\ \text{sem:} & \langle 2 \rangle :: \begin{bmatrix} \text{formula:} & \begin{bmatrix} \text{pred} : \text{sleeps} \\ \text{arg} : \langle 1 \rangle \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 22

Category of a one-place verb with construction rule.

⟨syn arg sem⟩ = ⟨sem formula arg⟩
⟨syn val sem⟩ = ⟨sem⟩

Figure 23

Direct construction rule in STUF.

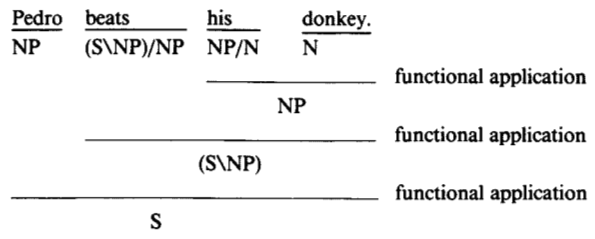


Figure 24

Functional-application analysis.

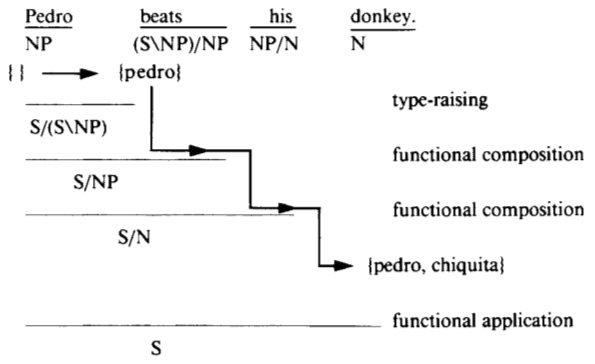


Figure 25

Left-to-right propagation of contextual information.

⟨sem formula in first⟩ = ⟨sem formula out first rest⟩
⟨sem formula in rest⟩ = ⟨sem formula out rest⟩
⟨sem formula out first first pred⟩ = beats
⟨sem first first arg1⟩ = ∅
⟨sem first first arg2⟩ = ∅

Figure 26

Partial DRS for a two-place verb (equational representation).

analysis in Figure 25 provides the antecedent *Pedro* when *his* is encountered. For backward anaphora, the information that an antecedent is expected would be passed through the "analysis tree."

As a paradigm for the treatment of contextual information, we want to sketch how anaphora resolution can be realized on Discourse Representation Structures (DRSs) [7] construed from an underlying CUG *in parallel* with syntax analysis.

First, a graph representation for (partial) DRSs must be chosen, illustrated in **Figures 26** and **27** by the graph representation of the two-place verb *beats*. In Kamp's notation this would correspond to **Figure 28.**

The graph representation for DRSs follows the proposals of Johnson and Klein [32] for organizing DRSs as list structures. Contrary to Kamp's initial notation, the list of reference markers and the list of conditions on reference markers are merged into one list (but markers are still distinct from conditions because they will have a specific representation; e.g., they could be values of a MARKER attribute).

Anaphora resolution can be expressed as a membership relation between an antecedent and a DRS, which is represented as a list of antecedents and conditions: $\langle(\text{rest}^* \text{ first})^+\rangle$. This membership constraint is stated in the semantic representation of the pronoun, as shown in **Figure 29**.

The values of the IN- and the OUT-attributes are used as place-holders in order to "thread" contextual information the DRSs contain through the syntax tree. In a CUG, "threading" is carried out by inheritance of attribute values among sister nodes. In the functional application rule, this means that FUNCTOR and ARGUMENT share substructures. With semantic construction rules reformulated for the construction of DRSs, the verb's semantic representation will fill the "scope" of its object. The representation of the whole VP is in the scope of the subject NP of the sentence. At this point, the information the pronoun is looking for is provided.

In a declarative formalism, looking for a member in a list means the same as predicting the existence of a member in a list. In this regard, the pure functional application approach of Figure 24 is equivalent to what appears in Figure 25. It seems, however, cognitively more appropriate to propagate information strictly from left to right during processing of a sentence. Categorial Unification Grammar, apparently, offers the means for experimentation with different syntacto-semantic approaches to model cognitive processing of language.

5. Conclusions

The advantages of unification-grammar formalisms for natural-language processing have been widely discussed in the literature. Among them are the ability to represent

partial information in an elegant way, the inherent potential for structure sharing, the declarative description of information flow, and a mathematically clean and computationally tractable type system with inheritance.

Our representation language supports all the desirable features of previous unification formalisms. Moreover, recent extensions to other formalisms such as disjunction and functional uncertainty are already integrated in its algebraic notation. In contrast to earlier formalisms, graphs are always built from other graphs by means of a small number of well-defined operations. Graphs can be embedded in graphs at any depth. At any place where a graph can occur in the specification of another graph, a graph name can be used as a place holder for a predefined graph.

The operation of graph application permits the encoding of a certain class of functions as graphs and the use of all complex graphs as functions from graphs to graphs. Here, the inherent partialness of graphs in a graph-unification system is exploited for the implementation of functional application with built-in parametrized polymorphism.

Although the formalism supports different brands of phrase-structure grammars, including the ID/LP notation of GPSG [1], it is especially well suited for lexicalized types of grammar such as Categorial Unification Grammar. The modularized specification of graphs permits the encoding of linguistic generalizations in a very concise way.

A graph-unification formalism like STUF also allows for the parallel processing of syntactic and semantic information. Lexical and contextual aspects of utterances are encoded using the same, declarative notation. Although different types and pieces of information about linguistic units are represented in a uniform way and as parts of the same structure, they can be kept apart conceptually and in the actual design of the grammar. They can be defined separately and combined either at compile time or at run time using the graph-name (template) facility of the formalism. The paralleling of syntactic rules (i.e., categories) and semantic construction rules meets the requirement of the principle of compositional semantics.

In this paper, we have shown examples of the use of STUF for the representation of linguistic knowledge. The application of the STUF language to the representation of the conceptual aspects of lexical semantics and to general knowledge representation are currently under investigation by our group.

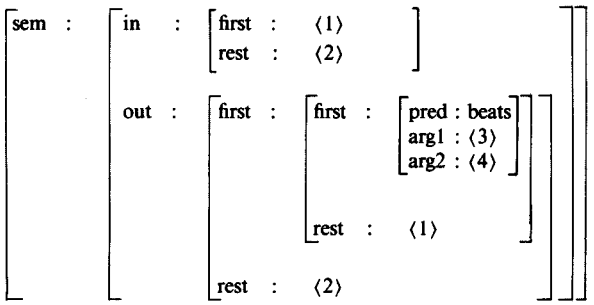


Figure 27

Partial DRS for a two-place verb (graph representation).

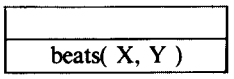


Figure 28

Sample DRS for two-place verbs in Kamp's notation.

⟨sem in (rest* first)$^+$ marker⟩
= ⟨sem distinguished_marker⟩

⟨sem scope in⟩ = ⟨sem in⟩
⟨sem scope out⟩ = ⟨sem out⟩

Figure 29

Semantic representation of anaphoric pronoun.

References

1. G. Gazdar, E. Klein, G. Pullum, and I. Sag, *Generalized Phrase Structure Grammar*, Blackwell, London, 1985.
2. *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed., MIT Press, Cambridge, MA, 1982.
3. G. E. Heidorn, "Automatic Programming Through Natural Language Dialogue: A Survey," *IBM J. Res. Develop.* **20**, 302–313 (1976).
4. C. Pollard, "Categorial Grammar and Phrase Structure Grammar: An Excursus on the Syntax–Semantics Frontier," *Proceedings of the Conference on Categorial Grammar* (*Studies in Linguistics and Philosophy*), E. Bach, R. T. Oehrle, and D. Wheeler, Eds., Tucson, AZ, May 31–June 2, 1985, Riedel, Dordrecht, the Netherlands, 1987.
5. K. Ajdukiewicz, "Die Syntaktische Konnexitaet," *Studia Philosophica* **1**, 1–27 (1935).

6. Hans Uszkoreit, "Categorial Unification Grammars," *Proc. COLING '86 (Proceedings of the 11th International Conference on Computational Linguistics)*, Bonn, West Germany, August 25–29, 1986, pp. 187–194.
7. Hans Kamp, "A Theory of Truth and Semantic Representation," *Formal Methods in the Study of Language*, J. A. G. Groenendijk et al., Eds., Mathematical Centre Tracts, Amsterdam, 1981, pp. 277–322.
8. I. Heim, "File Change Semantics and the Familiarity Theory of Definiteness," *Philosophy* **5**, 3–22 (1982).
9. M. Kay, "Parsing in Functional Unification Grammar," *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds., Cambridge University Press, Cambridge, 1985.
10. S. Shieber, H. Uszkoreit, F. Pereira, J. Robinson, and M. Tyson, "The Formalism and Implementation of PATR-II," Research on Interactive Acquisition and Use of Knowledge, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1983.
11. Fernando C. N. Pereira and Stuart M. Shieber, "The Semantics of Grammar Formalisms Seen as Computer Languages," *Proc. COLING '84*, Stanford, CA, July 2–6, 1984, pp. 123–129.
12. Hans Uszkoreit, "Syntaktische und semantische Generalisierungen im strukturierten Lexikon," *Proceedings of the German Workshop on Artificial Intelligence*, C.-R. Rollinger and W. Horn, Eds., Springer-Verlag, New York, 1986.
13. R. Kasper and W. Rounds, "A Logical Semantics for Feature Structures," *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Morristown, NJ, 1986, pp. 257–266.
14. Hans Uszkoreit, "STUF: A Description of the Stuttgart Unification Formalism," *LILOG Report 16*, IBM Germany, Science and Technology, D-7000 Stuttgart 80, 1987.
15. Lauri Karttunen, "Features and Values," *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, CA, July 2–7, 1984, pp. 28–33.
16. A. Ades and M. Steedman, "On the Order of Words," *Linguist. & Philos.* **4**, 517–558 (1982).
17. M. Flynn, "A Categorial Theory of Structure Building," *Order, Concord and Constituency*, G. Gazdar, E. Klein, and G. K. Pullum, Eds., Foris, Dordrecht, the Netherlands, 1983.
18. M. Steedman, "Dependency and Coordination in the Grammar of Dutch and English," *Language* **61**, 523–568 (1985).
19. D. Dowty, "Type-Raising, Functional Composition, and Non Constituent Coordination," *Proceedings of the Conference on Categorial Grammar (Studies in Linguistics and Philosophy)*, E. Bach, R. T. Oehrle, and D. Wheeler, Eds., Tucson, AZ, May 31–June 2, 1985, Reidel, Dordrecht, the Netherlands, 1987.
20. Mark Steedman, "A Categorial Theory of Intersecting Dependencies in Dutch Infinitival Complements," *Sentential Complementation, Proceedings of the International Conference held at UFSAL*, W. de Geest and Y. Putseys, Eds., Brussels, June 1983.
21. J. Hoeksema, "Categorial Morphology," Doctoral Dissertation, Rijks Universiteit Groningen, the Netherlands, 1984.
22. L. Karttunen, "Radical Lexicalism," presented at the Conference on Alternative Conceptions of Phrase Structure, New York, July 1986.
23. H. Zeevat, E. Klein, and J. Calder, "Unification Categorial Grammar," *Working Papers in Cognitive Science*, Vol. 1, Nicholas Haddock, Ewan Klein, and Glyn Morill, Eds., Centre for Cognitive Science, University of Edinburgh, Scotland, 1987.
24. G. Bouma, "Modifiers and Specifiers in Categorial Unification Grammar," *Linguistics* **26**, 21–46 (1988).
25. E. Bach, "On the Relationship Between Word-Grammar and Phrase-Grammar," *Nat. Lang. & Linguist. Theor.* **1**, 65–89 (1983).
26. E. Bach, "Generalized Categorial Grammars and the English Auxiliary," *Linguistic Categories*, Vol. 2, F. Heny and B. Richards, Eds., Reidel, Dordrecht, the Netherlands, 1983.
27. T. Vennemann and R. Harlow, "Categorial Grammar and Consistent Basic VX Serialization," *Theoret. Linguist.* **4**, 227–254 (1977).
28. A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA, 1985.
29. Emmon Bach, "An Extension of Classical Transformational Grammar," *Problems in Linguistic Metatheory, Proceedings of the 1976 Conference at Michigan State University*, 1976, pp. 183–224.
30. Esther König, "Methoden der Semantikkonstruktion in Unifikationsgrammatiken," *M.A. Thesis No. 466*, Institute of Computer Science, University of Stuttgart, West Germany, 1987.
31. R. Pareschi and M. Steedman, "A Lazy Way to Chart-Parse with Categorial Grammars," *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*, Stanford, CA, 1987, pp. 81–88.
32. Mark Johnson and Ewan Klein, "Discourse, Anaphora and Parsing," *Proc. COLING '86*, Bonn, West Germany, August 25–29, 1986, pp. 669–675.

Received April 20, 1987; accepted for publication August 6, 1987

Gosse Bouma *University of Stuttgart, Institute of Computational Linguistics, Keplerstrasse 17, D-7000 Stuttgart 1, West Germany.* Mr. Bouma is a research associate with the LILOG project at the University of Stuttgart. He studied general linguistics and Dutch at the University of Groningen. There he received both an M.A. degree in Dutch and an M.A. in linguistics in 1986. Mr. Bouma is currently working on categorial unification grammars.

Esther König *IBM Deutschland GmbH, Science and Technology – LILOG, P.O. Box 80 08 80, D-7000 Stuttgart 80, West Germany.* Ms. König is a Ph.D. candidate at the University of Stuttgart. She is currently working on the LILOG project of IBM Germany. Ms. König studied computer science and linguistics at the University of Stuttgart. Her thesis for the diploma in computer science was titled "Methods for the Construction of Semantic Representations in Unification Grammars." Ms. König is currently working on the theoretical and computational foundations of unification-based grammar formalisms.

Hans Uszkoreit *IBM Deutschland GmbH, Science and Technology – LILOG, P.O. Box 80 08 80, D-7000 Stuttgart 80, West Germany.* Dr. Uszkoreit is the project leader of the linguistic subproject of the LILOG project. He studied linguistics and computer science at the Technical University of Berlin and at the University of Texas at Austin. He received his Ph.D. from the University of Texas; his thesis was titled "German Word Order and Constituent Structure." From 1982 to 1986, Dr. Uszkoreit worked as a researcher with the AI Center at SRI International and with the Center for the Study of Language and Information at Stanford University. In 1986, he spent several months as a guest scientist with the LILOG project in Stuttgart.