UNIVERSITY OF PENNSYLVANIA
THE MOORE SCHOOL OF ELECTRICAL ENGINEERING
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

STRUCTURE UNIFICATION GRAMMAR: A UNIFYING FRAMEWORK FOR INVESTIGATING NATURAL LANGUAGE

James Henderson

Philadelphia, Pennsylvania
December 1990

A thesis presented to the Faculty of Engineering and Applied Science of the University of Pennsylvania in partial fulfillment of the requirements for the degree of Master of Science in Engineering for graduate work in Computer and Information Science.

Mitchell Marcus
(Advisor)

Mitchell Marcus
(Graduate Group Chair)

Abstract

This thesis presents Structure Unification Grammar and demonstrates its suitability as a framework for investigating natural language from a variety of perspectives. Structure Unification Grammar is a linguistic formalism which represents grammatical information as partial descriptions of phrase structure trees, and combines these descriptions by equating their phrase structure tree nodes. This process can be depicted by taking a set of transparencies which each contain a picture of a tree fragment, and overlaying them so they form a picture of a complete phrase structure tree. The nodes which overlap in the resulting picture are those which are equated. The flexibility with which information can be specified in the descriptions of trees and the generality of the combination operation allows a grammar writer or parser to specify exactly what is known where it is known. The specification of grammatical constraints is not restricted to any particular structural or informational domains. This property provides for a very perspicuous representation of grammatical information, and for the representations necessary for incremental parsing.

The perspicuity of SUG's representation is complemented by its high formal power. The formal power of SUG allows other linguistic formalisms to be expressed in it. By themselves these translations are not terribly interesting, but the perspicuity of SUG's representation often allows the central insights of the other investigations to be expressed perspicuously in SUG. Through this process it is possible to unify the insights from a diverse collection of investigations within a single framework, thus furthering our understanding of natural language as a whole. This thesis gives several examples of how insights from investigations into natural language can be captured in SUG. Since these investigations come from a variety of perspectives on natural language, these examples demonstrate that SUG can be used as a unifying framework for investigating natural language.

Acknowledgments

The work presented in this thesis has only been possible because of the diverse set of ideas and nondogmatic atmosphere provided by the computational linguistics group at the University of Pennsylvania. (I came, I saw, I concurred.) I would especially like to thank my advisor, Mitch Marcus, for his time, enthusiasm, and inspiration. Special thanks also go to Aravind Joshi and Mark Steedman for their ideas and comments, and to Bob Frank for making me defend my positions against seemingly all possible criticisms. I am also grateful to the CLiFF discussion group here at Penn for their comments and support. I'd also like to thank Amy Winarske, Yves Schabes, Tony Kroch, Ramesh Subrahmanyam, Anuj Dawar, and Michael Niv. This thesis is dedicated to my cats, Flotsam and Jetsam, who I'm sure will enjoy chewing on it.

Contents

Abstract ii

Acknowledgments iii

1 Introduction 1

2 Structure Unification Grammar 8

2.1 Describing Phrase Structure 9

2.1.1 The Notation 9

2.1.2 The Structure Models 11

2.1.3 The Descriptions 12

2.2 Accumulating Phrase Structure 14

2.2.1 Grammar Entries 15

2.2.2 Combining Structure Descriptions 16

2.2.3 Complete Structure Descriptions 17

2.2.4 The Derivations 18

2.3 A Formal Specification of SUG 20

2.4 Other Formalisms Using Partial Descriptions of Phrase Structure 22

3 Unifying Insights into Natural Language 24

3.1 Examples of Expressing Grammatical Constraints 25

3.1.1 Using SUG's Large Domain of Locality 25

3.1.2 Trading Ambiguity for Underspecification 30

3.1.3 Capturing Generalities 31

3.2 Lexical Functional Grammar 34

3.2.1 The Version of LFG 35

3.2.2 Expressing LFG′ in SUG 40
3.2.3 Discussion 56
3.3 Description Theory 57
3.4 Abney's Licensing Parser 59
3.4.1 The Parser 59
3.4.2 Comparison with SUG 62
3.5 Tree Adjoining Grammar 64
3.5.1 The Definition of TAG and FTAG 64
3.5.2 Expressing FTAG in SUG 66
3.5.3 Discussion 72
3.6 Lexicalized Tree Adjoining Grammar 73
3.7 Combinatory Categorial Grammar 77
3.7.1 Categorial Grammar with Token Identity and Partiality 78
3.7.2 CCG with Structure Categories 88
3.7.3 Capturing Coordination in SCCG 95

4 Conclusions and Future Directions **104**
4.1 Future Directions 107

Bibliography **112**

Chapter 1

Introduction

The study of natural language has yielded many insights. These insights have come from a diverse collection of investigations, each with its own perspective on the phenomena. This diversity is reflected in the plethora of representations and formalizations these investigations have used. Although the formalizations are usually incompatible, often the key insights of each investigation are not. Thus it should be possible to unify the insights from a variety of investigations within a single formalism. By investigating all these insights within a common framework, we can gain a better understanding of language as a whole. This thesis proposes that Structure Unification Grammar is an appropriate framework for such an investigation.

The key to finding such a framework is to extract the features common, or at least compatible, with all the formalisms. At first glance it appears we are left with nothing. However, there are a few characteristics which have been consistently useful. The first is the use of phrase structure. Some notion of phrase structure has been essential to almost every modern theory of language. The second characteristic is the use of partial descriptions to allow information to be accumulated over the derivation or parse. This eliminates the need to completely specify an entity as soon as it is introduced. The use of partial descriptions has been especially useful for theories which address computational issues, because it allows decisions to be delayed until more is known about the sentence. These two characteristics should be included in any formalism which attempts to perspicuously express insights from the wide variety of linguistic investigations. Thus the unifying framework should perspicuously represent phrase structure trees and should support the partial specification of this information. Structure Unification Grammar (SUG) is just such a framework; it is simply

a formalization of accumulating information about the phrase structure of a sentence until this structure is completely described.

Although many formalisms exist which can be viewed as constructing phrase structure trees from partial specifications, none allow the flexible specification of partial information in the way that SUG does. Like many other formalisms, SUG uses feature structures to allow the partial specification of node labels. For example, 'she' is nominative case, but 'Barbie' is ambiguous as to its case. Rather than giving 'Barbie' a different grammar entry for each possible case, the entry for 'Barbie' can simply not specify the case. Unlike most other formalisms, SUG also allows the specification of the structural relations to be equally partial. For example, if a grammar entry says a node with category S can have a child with category NP and a child with category VP, this does not preclude the same S node from also having other children, such as sentential modifiers. Also, grammar entries can partially specify ordering constraints between nodes, thus allowing for variations in word order. This ability to partially specify structural relations is extended in SUG with the addition of the dominance relation. Dominance is the recursive, transitive closure of the parent–child relation, here called immediate dominance. Among other things, this allows a grammar entry to specify that a trace NP is somewhere within an S, without specifying exactly where, thus expressing a long distance dependency within a local domain.

In SUG the source of these partial descriptions is the grammar. Each SUG grammar entry simply specifies an allowable grouping of information. Any of the information in a grammar entry can be in a phrase structure description, as long as all its information is there. Because of the complete flexibility SUG provides for specifying phrase structure information, the grammar can state exactly what these information interdependencies are. This ability to say what you know where you know it will be crucial in the discussion of capturing grammatical constraints. Intuitively, each grammar entry can be depicted as the fragment of tree structure which it specifies. Any tree which is generated by a grammar can be depicted by overlaying these depictions of tree fragments.

A complete description of a phrase structure tree is constructed from the partial descriptions in an SUG grammar by conjoining a set of grammar entries and specifying how these descriptions overlap. The way two descriptions overlap is by sharing nodes. In other words, a set of descriptions can be combined by conjoining them and doing zero or more equations of pairs of their nodes. In the tree depiction given above, the overlaying of tree fragment depictions corresponds to the conjoining of the descriptions and the nodes which overlap

in the resulting picture are the ones which are equated. As should be obvious from this graphical representation, if node equations were not allowed the resulting description would not specify a complete tree. What equations are allowed is only restricted by the requirement that there be at least one phrase structure tree which is compatible with the resulting description. An example of combining descriptions is given in figure 1. One description specifies the immediate children of S, another the structure of the NP "Barbie", and the third the structure of the VP "poses". By conjoining these descriptions and doing the two equations shown with circles, we produce a complete description of the phrase structure tree for the sentence "Barbie poses". By using this very general combination operation, the structure of a derivation is in no way restricted by the structures used in the grammar. This flexibility is crucial for unifying within a single framework the insights from both grammatical investigations of language and more procedural investigations of language.

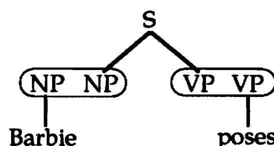


Figure 1: An example of combining structure descriptions. The circled nodes are equated.

When all the information about a phrase structure tree has been accumulated, the resulting description should completely specify that phrase structure tree. However, since the result of a derivation is a partial description, there are always an infinite number of trees which satisfy it. In order to make this partial description a complete description, SUG assumes that anything which is not entailed by the description is false. For some descriptions this will work, because they specify all and only the positive information in some phrase structure tree. These descriptions are called complete descriptions of this phrase structure tree, as was mentioned for the phrase structure of "Barbie poses" in the previous paragraph. For other descriptions this assumption makes them unsatisfiable by any phrase structure tree. For example, if the description specifies that a given terminal exists but does not specify what its word is, then this terminal will be assumed not to have any word. Such an assumption will make the description unsatisfiable, since all terminals have, possibly empty, words. The implication of this is that only derivations which result in complete descriptions are valid SUG derivations.

Despite the simplicity of this system, SUG is extremely powerful. Without restrictions

on the use of feature structures it has Turing Machine power. Even when these feature structures are restricted to being atomic, SUG is strictly more powerful than Tree Adjoining Grammar, and can generate the language $a_1^n a_2^n \ldots a_m^n$, for any fixed m[1]. Unlike many computational models with this power, SUG provides a perspicuous representation for investigating natural language. It is the combination of this power and this perspicuity which makes SUG a suitable framework for unifying a diverse collection of investigations into the nature of language.

The perspicuity of SUG's representation of grammatical information comes from three major characteristics. The first is SUG's ability to partially specify information. This permits a grammar entry to say as much and only as much as is desired. The second characteristic is SUG's large domain of locality for specifying grammatical information. Most importantly, both long distance dependencies and predicate–argument relationships can be stated directly within single grammar entries, without the need to pass this information through special node label features. The third characteristic is that there is no limit on the amount or kind of overlap between grammar entries in the derived structure. Thus two separate grammar entries can add constraints to the same set of nodes. This allows grammatical information to be separated according to information dependencies rather than according to structural configurations. Each of these characteristics are important in SUG's ability to perspicuously express the variety of grammatical constraints found in the formalisms discussed here.

The first formalism discussed here is Lexical Functional Grammar. LFG has a very expressive language for specifying grammatical constraints, and an explicit representation of semantic information which also constrains the possible derivations. SUG is sufficiently expressive to specify almost all the constraints specifiable in LFG. This includes the ability to constrain possible long distance dependencies, and the ability to express LFG's representation of semantic information in the feature structure labels of SUG nodes.

The second investigation discussed is Description Theory. D-Theory makes extensive use of the partial specification of phrase structure information in order to do syntactic parsing incrementally and deterministically. Partial specifications allow a D-Theory parser to only

[1]This power means that SUG in its pure form can not be parsed very efficiently. I am not addressing this issue in this paper because here I am only concerned with demonstrating the perspicuity and power of this simple system. Presumably the subset of this power which is actually needed to parse natural languages is quite efficiently parsable. How to characterize this subset is the topic of my current research. This will be mentioned at the end of this thesis in my discussion of future research directions.

specify what it is sure of, delaying the specification of other phrase structure information until later in the parse. SUG's use of partial specifications allow for the same degree of flexibility, thus also supporting an incremental deterministic parser.

Another place where parsing considerations have crossed successfully with linguistic investigations is Abney's licensing parser. Abney extends the linguistic notion of licensing so that all phrases must be licensed, and parses sentences by inferring their licensing relations. These licensing relations are both very general across languages and, when represented properly, can easily be recovered by a psychologically plausible parser. One interesting aspect of Abney's representation of these relations is the need for anti-relations, which are specified with the licensee rather than with the licensor. Anti-relations are used primarily for licensing adjuncts. The close relationship between licensing relations and phrase structure relations permits SUG to manifest the same information in its representation of phrase structure. Because the division of grammatical information in SUG does not have to follow any specific structural configurations, both regular licensing relations and anti-relations can be supported. Thus SUG also supports an efficient psychologically plausible parser for recovering licensing information.

Tree Adjoining Grammar is also discussed in this thesis. Like SUG, the data structures of a TAG grammar are phrase structure trees. However, the combination operation of TAG, adjunction, is quite different from that of SUG. TAG still has a large domain of locality for specifying grammatical constraints. It can state both long distance dependencies and predicate–argument relationships directly within single grammar entries, as was discussed for SUG. Linguistic work in TAG (for example [Kroch and Joshi, 1985]) has pointed out the importance of these abilities. The explicit representation of phrase structure in TAG, and SUG, is also useful because it provides for a distinction between phrase structure and derivation structure, which will be important in combining the insights of CCG with those of TAG analyses and other linguistic work.

Lexicalized Tree Adjoining Grammar adds the operation of substitution to TAG, thereby permitting TAG grammars to be expressed lexically. This addition greatly increases the flexibility with which information can be divided among grammar entries, thus permitting lexicalization, but LTAG is still less flexible than SUG in this regard. These constraints may be desirable linguistically, but it appears they can be manifested in SUG grammars if desired. The use of an explicit representation of phrase structure in both LTAG and SUG, and SUG's ability to express the information dependencies expressible in LTAG, allow SUG

to use the same analyses as LTAG in the specification of a lexicalized grammar.

The last investigation discussed is Combinatory Categorial Grammar. CCG proposes a notion of constituent structure which is much different from the semantically based conception used in the above investigations. CCG's constituent structure is motivated by coordination and extraction phenomena. The data structures in CCG are curried functional types, and the primary combination operations are function application and function composition. A phrase is a constituent if the types from each word in the phrase can be combined into a single type. Two constituents can coordinate if they can each be reduced to the same type, with the result of the coordination being that type. This approach allows what is usually called nonconstituent coordination to be treated as constituent coordination. This approach does a very good job at handling coordination phenomena, but it lacks the perspicuous representation provided by a system like LTAG. SUG bridges this gap by providing structures which both have explicit phrase structure, like LTAG trees, and behave like functional types, like CCG types. By interpreting these SUG structures as the functional types they simulate, CCG's analysis of coordination can be applied to SUG's representations. In this way the important characteristics of CCG's constituent structure can be captured within SUG's derivation structure, while still expressing conventional phrase structure within SUG's explicit representation of phrase structure.

Although this thesis is primarily concerned with the representation of grammatical information, there are also reasons to believe that SUG provides a good representation for processing that information. As is pointed out in the discussion of D-Theory and Abney's licensing parser, the partiality and flexibility of SUG's representation supports parsers for natural language which are incremental, deterministic, and have other psychologically plausible characteristics. The work on incorporating CCG's notion of functional types into SUG provides another tool which is of great interest in developing psychological models of language processing. These types provide a theory of how certain information in structures can be abstracted away from, thus allowing many otherwise arbitrarily large structures to be represented in bounded memory. This allows the investigation of parsers which have bounds on the size of their memory, thus also bounding the amount of computation necessary to parse. This later work will be discussed at the end of this thesis in the section on future research.

The remainder of this thesis will define SUG more precisely, and show how it captures the insights from various investigations into the grammatical and computational nature of

language. Chapter 2 starts with an extended discussion and definition of SUG, including its formal specification. The last section of chapter 2 then compares SUG to other formalisms which have addressed the issue of the partial specification of phrase structure. Chapter 3 discusses the above investigations into language and how their insights can be unified within SUG. The first section of this chapter gives examples of how to perspicuously express a variety of grammatical constraints in SUG. The other sections discuss Lexical Functional Grammar, Description Theory, Abney's licensing parser, Tree Adjoining Grammar, Lexical Tree Adjoining Grammar, and Combinatory Categorial Grammar. This thesis ends with some concluding remarks and a discussion of future research directions.

Chapter 2

Structure Unification Grammar

As discussed in the introduction, Structure Unification Grammar is a formalization of accumulating information about the phrase structure of a sentence until this structure is completely described. This chapter will expand the description given in the introduction by giving the details of SUG's definition. It will also compare SUG with other formalisms based on partially specifying phrase structure. The subsequent chapter will show how SUG can unify the insights from a variety of investigations into natural language.

The first section in this chapter discusses the language which SUG uses to describe phrase structure trees. These trees are ordered trees of feature structures. The tree relations are immediate dominance, linear precedence, and dominance. Immediate dominance is the relationship between a node and each of its immediate children. Linear precedence is the ordering relation used here. Dominance is the recursive transitive closure of immediate dominance. Its addition is necessary in order to express long distance dependencies in a single grammar entry. The nodes of the trees are feature structures. They are divided into nonterminals, which are arbitrary feature structures, and terminals, which are atomic instances of strings. These feature structures are allowed to share values, including having the value of a feature be another node. For example, a node may have a feature *head* whose value is one of the node's children. More examples of how this descriptive language is used to express grammatical information are given below and in section 3.1.

The second section in this chapter specifies what constitutes an SUG derivation. The objects used in these derivations are partial descriptions in SUG's language for specifying phrase structure trees. Each step in a derivation combines descriptions by conjoining them and adding zero or more statements of equality between nonterminal nodes in the

descriptions, under the condition that the resulting description is satisfiable. The leaves of a derivation tree must be entries from the grammar, and the root must be a complete description. A description is complete if assuming that anything which is not entailed by the description is false, makes the description satisfied by a unique phrase structure tree. The tree set generated by a grammar is the set of trees specified in this way by some description which is the result of some derivation for the grammar. The language generated by a grammar is the yields of these trees. Examples of each of these definitions will be given below.

To make the definition of SUG precise, the third section in this chapter gives a concise formal specification of SUG. The reader may want to skip this section.

The last section in this chapter discusses how SUG compares to other formalisms which can be viewed as using partial descriptions of phrase structure. The formalisms I will discuss are CFGs, PATR-II, and the formalization of FUG given in [Rounds and Manaster–Ramer, 1987]. Some other formalisms which can be viewed in this way will be discussed in chapter 3.

2.1 Describing Phrase Structure

The central concept in Structure Unification Grammar is the partial description of phrase structure. It allows for great flexibility in both the specification of grammatical information and the processing of that information. This section presents the language which SUG uses to describe phrase structure.

2.1.1 The Notation

In recent years many linguistic formalisms have been developed which use partial descriptions of linguistic information. These formalisms usually use feature structures to represent this information. The problem with feature structures is that the relationships which they can represent are restricted to being functional, in the sense that a feature structure label must represent a function from feature structures to feature structures. This causes trouble when specifying information about phrase structure, since many of the relations which we wish to state, such as linear precedence and dominance, are not functions. Formalisms like PATR-II ([Shieber, 1986]) solve this problem by using a separate mechanism for specifying phrase structure. PATR-II uses a context free skeleton for this purpose. Description Theory ([Marcus *et al.*, 1983]) takes a different approach. It extends feature structures to

allow structural relations to be expressed in the same manner as the information usually expressed in feature structures[1]. This later approach gives the description of the phrase structure the same degree of partiality given the other information. For this reason this is the approach which will be taken here.

There have been several suggestions for how to add arbitrary relations to feature structures. One was proposed in [Rounds, 1988], where set values are added to feature structures. This would allow linear precedence, for example, to be expressed by giving a node a feature with a set value containing all the nodes which precede it. However, this approach would force an unwanted asymmetry in the representation between preceding and being preceded by. Instead I will not use the automata based conception of feature structures used by Rounds, but use a representation espoused by Johnson in [Johnson, 1990]. In this representation feature structures are specified using quantifier-free first-order formulae with equality. In these formulae, variables range over feature structures, atoms are represented as constants, and labels are specified as unary functions from feature structures to feature structures. In [Johnson, 1990], the characteristics of atoms and a treatment of incomplete information are axiomatized. This axiomatization will be discussed in section 2.1.3. The advantage of this system over Rounds' representation of feature structures is that quantifier-free formulae already have a mechanism for specifying arbitrary relations, namely predicates. For example, if node x precedes node y this can be expressed as *precedes(x,y)*[2].

The shift to using quantifier-free formulae as the notation for feature structures suggests a few changes which I will adopt. Since a typical formula will contain many variables, none of them distinguished from the others, I will treat a formula as describing a set of entities, rather than a single one. This has the consequence that our phrase structure descriptions no longer need to be root centered. Given that we are talking about sets of entities, it is also natural to remove the restriction that they all be connected.

[1] Rounds and Manaster-Ramer take a similar approach in [Rounds and Manaster–Ramer, 1987]. This will be discussed in section 2.4.

[2] The problem with Johnson's representation of feature structures is that he uses the usual classical semantics for first-order formulae. This means that, unlike in Rounds' system, in his system subsumption does not respect entailment, where subsumption is as defined in [Rounds and Kasper, 1986]. In other words, given two feature structure models, A and B, such that the nonnegative information in A is a subset of that in B (A subsumes B), there may be descriptions which are satisfied by A but not by B. This is because a description may have a negative constraint which is incompatible with information which is in B but not in A. This will not be a problem here because the use of negation is limited to axioms in the definition of SUG which either are true in all phrase structure tree models, or are simply predicating something's existence. Thus this problem can not arise, and in SUG subsumption does respect entailment, with the models restricted to those specified in the next section.

First-order formulae not only provide us with a natural representation for our descriptions, they also provide a way to axiomatize the characteristics of the relations we wish to add. Stating relations between nodes will have no causal role in a parse if we do not restrict these relations in accordance with their intended meaning. These axioms can simply be added to the set already introduced by Johnson to define the nature of atoms and undefined information. In order to do this the notation will have to be expanded to first-order formulae with quantifiers. The only problem with this is that the satisfiability problem for first-order formulae with quantifiers is undecidable. However, we already know that SUG is in general undecidable. Quantifiers will still be excluded from grammar entries.

2.1.2 The Structure Models

Before discussing how to describe phrase structure, it is necessary to specify the objects to be described. I will restrict the set of models for the descriptions to ordered trees of feature structures. The nodes of these trees are divided into two types, terminals and nonterminals. The nonterminals are models of arbitrary feature structures[3]. Terminals are all instances of strings. The terminals must be instances of strings rather than strings because otherwise the phrase structure of a sentence with the same word occurring twice would not be a tree. Values in the feature structures can corefer, both within a single node and between the feature structures for different nodes. This includes the ability to have a node be the value of a feature in another node.

The only components of the allowable structures other than the above feature structures are the two ordered tree relations, immediate dominance and linear precedence. Immediate dominance is the relationship between a node and each of its immediate children. The graph of the immediate dominance relation must be a single tree. Linear precedence is the ordering relation. It is a partial order on nodes which is transitive and antisymmetric. Also, if a node x linearly precedes a node y, then everything in the subtree below x linearly precedes everything in the subtree below y[4].

[3]Any models of simple feature structures will do here, as long as they must be single feature structures and must be connected. One such set of models is given in [Rounds and Kasper, 1986].

[4]There are a couple other constraints which could be imposed on the allowable models, which I have not chosen to include. One is that the root of the tree have category S, but this seems better incorporated at the level of a linguistic theory. Another is that the linear precedence relations completely order the terminals, since the words of a sentence are always completely ordered in either time or space. I have not included this constraint because there seem to be sentences in some languages for which some of this ordering is not significant to the sentence's phrase structure.

2.1.3 The Descriptions

As discussed above, the language SUG uses to describe models of phrase structure trees uses first-order logic as its notation. In this representation variables range over feature structures and the constant $\perp$, constants represent atomic feature structures, unary function symbols and equality are used to represent feature–value relationships, and predicates are used to represent tree relations. [Johnson, 1990] shows how to represent the feature structures in this way. If a feature structure x has y as its f feature's value, this is represented as the statement $f(x) \approx y$. The constant $\perp$ is used to represent nonexistent values of functions, since first-order logic requires functions to be total[5]. The use of functions to specify feature values enforces the fact that a given feature structure can have only one value for each of its features. The characteristics of constants are enforced with the following axioms, taken from [Johnson, 1990].

1. For all constants c and feature labels f, $f(c)=\perp$
2. For all distinct pairs of constants c_1 and c_2, $\neg(c_1=c_2)$

The characteristics of $\perp$, which represents nonexistent information, are axiomatized as follows, also taken from [Johnson, 1990].

3. For all feature labels f, $f(\perp)=\perp$
4. For all constants c, $\neg(c=\perp)$

Finally, when the value of a feature is specified then it must exist. This means that the specification can not be done simply using equation, since $f(x)=y$ is consistent with $y=\perp$. Thus Johnson defines another operator "$\approx$" to be used for specifying features, which is defined as follows.

For all terms u, v, $u \approx v \Leftrightarrow (u=v \wedge \neg(u=\perp))$

This is not an axiom, since there are an infinite number of instantiations of it, but a definition of what $\approx$ is an abbreviation for.

The axiomatization of the tree relations are done similarly to the above axioms, only tree relations are specified using predicates rather than functions. The predicates *idom* and *prec*

[5] Johnson says this symbol is for undefined information, but I will use the term "nonexistent" because it is less easily confused with the term "unspecified". A feature structure can be completely unspecified and yet still exist.

specify immediate dominance and linear precedence relations between nodes, respectively. Formulae may also specify dominance relations between nodes using the predicate *dom*. Dominance is the recursive, transitive closure of immediate dominance. Thus a node x dominates a node y either if x equals y or if there are a series of nodes z_1 to z_n such that x equals z_1, y equals z_n, and for all i between 1 and $n-1$, z_i immediately dominates z_{i+1}. Nodes are distinguished from other feature structures using the predicate *node*, and terminals are distinguished from nonterminals using the predicate *terminal*. These predicates are axiomatized as follows, where *strings* is the set of instances of strings.

$\forall\ x,y,z,w,$

5. $\neg(idom(x,x))$
6. $idom(x,y) \wedge idom(z,y) \Rightarrow x \approx z$
7. $idom(x,y) \Rightarrow dom(x,y)$
8. $\neg(prec(x,x))$
9. $prec(x,y) \wedge prec(y,z) \Rightarrow prec(x,z)$
10. $prec(x,y) \wedge dom(x,w) \wedge dom(y,z) \Rightarrow prec(w,z)$
11. $dom(x,y) \wedge dom(y,z) \Rightarrow dom(x,z)$
12. $dom(x,y) \wedge dom(y,x) \Rightarrow x \approx y$
13. $dom(x,y) \Rightarrow (x \approx y \vee \exists z(idom(z,y) \wedge dom(x,z)))$
14. (a) $prec(x,y) \Rightarrow (node(x) \wedge node(y))$
    (b) $dom(x,y) \Rightarrow (node(x) \wedge node(y))$
    (c) $terminal(x) \Rightarrow node(x)$
15. $node(x) \Rightarrow dom(x,x)$
16. $terminal(x) \Leftrightarrow (\exists s \in strings,\ x \approx s)$
17. $terminal(x) \wedge dom(x,y) \Rightarrow x \approx y$

Figure 2 gives an example of how phrase structure is specified in this descriptive language. Not all the information about the structure is explicitly specified in the formula, but the rest is derivable given the above axioms.

The above axioms complete the definition of the language SUG uses to describe phrase structure trees. The grammar specifications can only use a subset of the expressive power

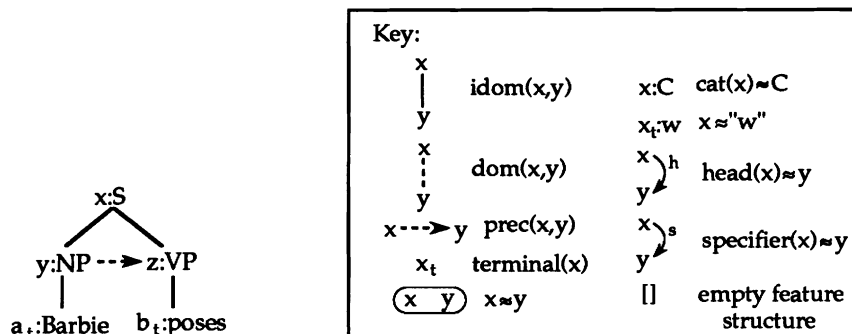


Figure 2: The structure specified by $(cat(x) \approx S \wedge cat(y) \approx NP \wedge cat(z) \approx VP \wedge a \approx$ *"Barbie"* $\wedge\ b \approx$ *"poses"* $\wedge\ idom(x,y) \wedge idom(x,z) \wedge idom(y,a) \wedge idom(z,b) \wedge prec(y,z))$. The key to the right defines the symbols I will use to graphically depict phrase structure information. When the variable names are not important I will simply specify the category or word of the node.

of this language, but the full power is necessary in order to express and reason with the axioms. In particular, all the variables in a grammar entry must be existentially quantified and the only logical connective which can be used is conjunction; universal quantification, disjunction, and negation cannot be used. This will be discussed more in the following section.

2.2 Accumulating Phrase Structure

With the above language for describing phrase structure it is now possible to define the process of deriving phrase structure trees in Structure Unification Grammar. An SUG derivation starts with partial descriptions of phrase structure from the grammar, and sticks them together using node equations, until a complete description of some phrase structure tree is constructed. That tree is the result of the derivation. This process can be visualized as taking a set of transparencies, each with a grammar entry on it, and placing them on top of each other[6] until the resulting picture is of a complete phrase structure tree[7]. As

[6]Of course, this will only work if the original transparencies have their information spatially laid out in a way compatible with the total resulting picture.

[7]This characterization is slightly misleading, since there will be information about the resulting description, as a consequence of the axioms, which is not depicted in any of the original transparencies. New dominance and precedence relationships between nodes are an obvious example of this, although there are

this depiction implies, the descriptions in the grammar are not arbitrary formulae in the language for describing phrase structure. This would allow negative facts, disjunctive facts, and universal facts to be expressed in the grammar entries, all of which cannot be depicted in this simple way. Grammar entries are restricted to being conjunctions of facts with only existentially quantified variables. The restrictions on what is a complete description of a phrase structure tree are defined by the need to have a unique phrase structure tree as the result of the derivation. As with any partial description, the description resulting from a derivation has an infinite number of phrase structure trees which are compatible with it. One way to find a unique tree for a description is to assume that anything which is not entailed by this description is false. This definition can only find such a tree for a subset of the descriptions, called complete descriptions. All descriptions in this subset must specify the immediate parent of every node except the root, and must specify the string associated with every terminal. This section will go into the above discussion in more detail.

2.2.1 Grammar Entries

An SUG grammar simply consists of a set of partial descriptions of phrase structure. These descriptions specify what configurations of information are allowed by the grammar. If a particular description is in the grammar, then that description's information can be added to a description in a derivation, as long as all its information is added and the result is satisfiable. For example, the grammar entry $(cat(x) \approx S \wedge cat(y) \approx NP \wedge cat(z) \approx VP \wedge idom(x,y) \wedge idom(x,z) \wedge prec(y,z))$ allows two nodes whose *cat* features are compatible with NP and VP, respectively, to attach under a node with a *cat* feature compatible with S, but in the resulting description the NP node must precede the VP node. This example could equally well be described with the precedence information being the precondition and the category information being the result, but regardless the requirement is the same; all the information can be included as long as all the information is included. Other examples will be given throughout the rest of this thesis. This meaning of grammar entries may be clearer in the case of a lexicalized grammar. In this case the presence of a word in a sentence can "license" the portion of the complete structure which is specified in one of the word's grammar entries, as long as the rest of the structure is compatible with this portion.

other less obvious possibilities. Nonetheless, all the information about the resulting description can be recovered from the resulting depiction using the axioms. In any case, this characterization is a useful way to think about SUG derivations.

The entries in an SUG grammar are not arbitrary partial descriptions of phrase structure. They are restricted to a subset of SUG's language for describing phrase structure. SUG grammar entries can only have existentially quantified variables and the only logical connective allowed is conjunction. They cannot use universally quantified variables, disjunction, or negation. Because all variables in a grammar entry are existentially quantified, the quantifiers are not explicitly specified. These restrictions are imposed for several reasons. First, they ensure that in SUG subsumption respects entailment. Second, they greatly simplify determining if a description is complete. If negation or disjunction were allowed in the grammar entries, then a grammar entry could specify grammar specific characteristics which need to be uniquely determined for the description to be complete[8]. Third, it restricts the domain of locality of grammar entries. If universal quantification was allowed in grammar entries then they could directly constrain nodes which are not mentioned in their description. Lastly, the intuitive characterization of SUG as simply constructing a picture of the derived phrase structure tree by overlaying pictures of the grammar entries, would not be possible without these restrictions on the language used to specify SUG grammar entries.

Grammar entries are the leaves of SUG derivation trees. However, if the same grammar entry is used twice in the same derivation, then the two instantiations of the grammar entry cannot be identical. First, the two instances must use disjoint sets of variables. This is simply a technique for avoiding variable capture during the derivation due to changing the scope of the implicit existential quantifiers. Second, the two instances must have distinct terminals. When the same word occurs twice in a sentence it must be manifested as two distinct terminals in the phrase structure, otherwise the phrase structure is not a tree. Thus whenever a grammar entry is introduced into a derivation, all its terminals are replaced with new unique instances of their words. This has the effect of preventing any two terminals with their words specified from equating.

2.2.2 Combining Structure Descriptions

The combination operation in SUG derivations is very simple. A set of descriptions are combined by conjoining them and adding zero or more statements of equality between

[8]Other than this complication there are no problems with allowing disjunction in grammar entries. Not permitting disjunction does not restrict the languages generatable by SUG, since any disjunction can be specified with a grammar entry for each possible choice in the disjunction.

pairs of their nonterminals. Simply taking the conjunction of the descriptions would not be sufficient, since the fragments would never become connected, and thus would never form a complete description of a tree. Permitting arbitrary information to be added would not permit the grammar to constrain the set of derivable phrase structure trees. By only allowing coreference information to be added SUG avoids both these problems, and it conforms to the intuitive characterization of SUG as simply constructing a picture of the derived phrase structure tree by overlaying pictures of the grammar entries. An example of this combination operation is given in figure 3. The only restriction on what equations can be added is that the resulting description be satisfiable. This is exactly analogous to unification in normal feature structures, which is also specified in this notation as equation under the condition that the result be satisfiable. It is worth noting that the set of equations used in combining two descriptions is not determined uniquely. The definition of a combination is nondeterministic. Descriptions S and T can combine to produce a satisfiable description U if there exists a conjunction of equations of nonterminals, E, such that $U = S \wedge T \wedge E$. Also note that the fact that only the equation of nonterminals can be added does not prevent terminals from equating, since the unification of features in nonterminals can cause the equation of terminals as a side effect.

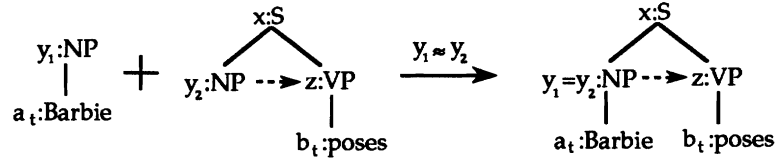


Figure 3: The combination of $(cat(y_1) \approx NP \wedge a \approx \text{“Barbie”} \wedge idom(y_1, a)) = F_1$ with $(cat(x) \approx S \wedge cat(y_2) \approx NP \wedge cat(z) \approx VP \wedge b \approx \text{“poses”} \wedge idom(x,y_2) \wedge idom(x,z) \wedge idom(z,b) \wedge prec(y_2,z)) = F_2$ using the equation $y_1 \approx y_2$ to form $(F_1 \wedge F_2 \wedge y_1 \approx y_2)$.

2.2.3 Complete Structure Descriptions

When a derivation is done there needs to be a phrase structure tree which it derives. However, the result of a derivation is a description, not a tree. The question is, what phrase structure tree does the resulting description specify? Given a partial description, the usual way to to make it a complete description is to invoke the closed world assumption. In other words, the description is assumed to specify everything which is true about the thing being described. Under this assumption, anything which is not entailed by the description is

false. However, this will not produce a satisfiable set of constraints if the original description contains disjunctive information. If the description entails $f \vee g$ but does not entail f and does not entail g, then this assumption will produce a description which entails $(f \vee g) \wedge \neg f \wedge \neg g$, which is unsatisfiable.

In SUG descriptions the above problem arises in two ways. First, if a node x is dominated by a distinct node and x does not have an immediate parent specified, then there is an ambiguity as to what the immediate parent of x is, as is manifested in axiom 13 in section 2.1.3. This ambiguity means that after applying the closed world assumption there will be no tree models which satisfy the description. In other words, for any description which has some nonroot node without its immediate parent specified, the closed world assumption will produce an unsatisfiable description. The other way this problem arises is when a terminal is specified to exist but no word is specified for it. When the closed world assumption is applied to such a description, the terminal will be assumed not to be equal to any instances of strings. Because in phrase structure tree models all terminals are instances of strings, no models will satisfy the resulting description. These facts imply the only way the closed world assumption will produce a satisfiable description is if all the terminals which are known to exist have their word specified and all nodes except the root have an immediate parent specified. Thus in order to use this method for determining the resulting phrase structure tree, the resulting description must have all the terminals' words specified and all the nonroot nodes' parents specified. In SUG such a description is called a complete description, because it completely specifies a unique phrase structure tree under the assumption that anything which is not entailed by the description is false.

The above approach to finding a unique phrase structure tree for a given description only works for complete descriptions. Since we do not want to make arbitrary choices when determining the resulting tree of a derivation, the only derivation trees which can be allowed are those which result in such a complete description. This is precisely the requirement for finished SUG derivations; the resulting description must be complete. Many other constraints on the resulting descriptions of finished derivations can be enforced in the grammar using features and underspecified terminals, as will be demonstrated in chapter 3.

2.2.4 The Derivations

As mentioned above, an SUG derivation starts with descriptions taken from the grammar, combines them by conjoining them and adding equations between nodes, and ends with a

complete description which specifies the resulting tree of the parse. Each of these components of a derivation are discussed at length in the previous sections. Such a derivation can be described as a tree, the leaves of which are the initial descriptions, the internal nodes of which are the intermediate descriptions, and the root of which is the resulting description. All the descriptions in an SUG derivation tree must be satisfiable, otherwise the resulting description would also be unsatisfiable. The leaves of an SUG derivation tree are entries from the grammar, except their variables have been replaced with fresh variables and their instances of strings have been replaced with fresh instances of strings. This replacement is done in such a way that all the leaves of a derivation tree have disjoint sets of variables and disjoint sets of instances of strings. This process is done to prevent two instantiations of the same grammar entry from getting their variables or terminals unintentionally conflated. Each internal node of an SUG derivation tree is the conjunction of its children, plus a conjunction of zero or more equations between nonterminal nodes in its children. Because each description in a derivation must be satisfiable, the sets of equations are limited to those which result in satisfiable descriptions. There are no other restrictions on these equations. The root of an SUG derivation tree must be a complete description. This means this description must specify an immediate parent for all its nonroot nodes, and must specify an instance of a string for all its terminals. This requirement guarantees that the resulting description will specify a unique phrase structure tree after assuming that anything not entailed by the description is false. This unique tree is the resulting tree of the derivation. The sentences whose words and ordering are compatible with the terminals of the resulting tree are the resulting sentences of the derivation. Note that there may be more than one such sentence, since the ordering of the terminals may be underspecified.

An example derivation is shown in figure 4. The leaves of the derivation tree are possible grammar entries for 'Ken', 'poses', and 'shamelessly', and are given at the top of the figure. The first step of the derivation combines the first two structure descriptions with the equation $y_1 \approx y_2$. The second step combines this structure description with that for 'shamelessly' with the equation $z_1 \approx z_2$, thus forming a complete description of the tree shown at the bottom of the derivation. The only sentence compatible with the ordering constraints on this resulting tree is "Ken poses shamelessly". Note that many other derivation structures are possible, including the one step derivation which combines all three structures with both equations at the same time. In fact, all derivations will have an equivalent derivation for each possible way of combining the grammar entries.

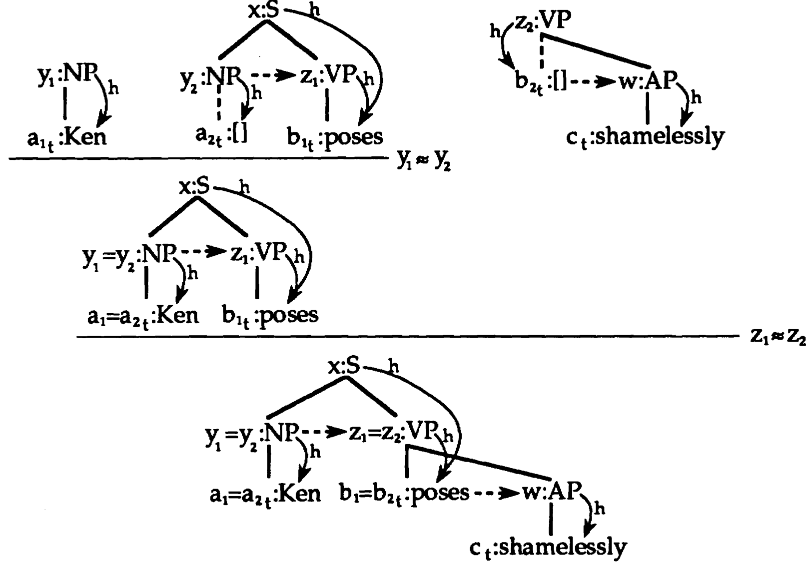


Figure 4: A derivation of the sentence "Ken poses shamelessly". The descriptions depicted in the top row are grammar entries and the tree depicted at the bottom is the result of the derivation. See figure 2 for an explanation of the notation used here and in subsequent figures.

2.3 A Formal Specification of SUG

To clarify the above discussion, the following is a formal specification of an SUG grammar and the sentences it generates. An SUG grammar is a tuple $\langle S, L, A, V\rangle$, where V is the variables, $A \cup$*strings* is the constants, L is the function symbols, {*idom, prec, dom, terminal, node*} is the predicates, and S is a finite set of first order formulae in these primitives. The formulae in S do not use disjunction or negation, and all their variables are implicitly existentially quantified. The arity of all functions is one. *Strings* is a set of instances of strings. The arities of *terminal,* and *node* are one. The arities of *idom, prec,* and *dom* are two. What satisfies a formulae and what a formulae entails are always determined with respect to the axioms given in section 2.1.3.

A description F is generated by a grammar $\langle S, L, A, V\rangle$ if and only if F is satisfiable, F is complete, and $F = F_1 \wedge \ldots \wedge F_n \wedge E$, where the variables and instances of strings in F_1 through F_n are disjoint, there exits a substitution θ for variables and instances of strings such that $F_1[\theta],\ldots,F_n[\theta] \in S$, and E is a conjunction of equations between nonterminals in F. A formula F is complete if for every terminal x in F, F entails $x \approx s$ where s is an instance

of a string, and for every node x in F, F either entails $x{\approx}r$, or there is a node y such that F entails $idom(y,x)$, where r is a unique node in F. x is a terminal in F if F entails $terminal(x)$, x is a node in F if F entails $node(x)$, and x is a nonterminal in F if x is a node in F but not a terminal in F.

A tree is generated by a grammar if it is the subsumption minimal phrase structure tree for some description generated by the grammar. A tree T is the subsumption minimal phrase structure tree for a description F if T satisfies F, and, for all trees T' which satisfy F, T subsumes T'. Such a tree will always exist and be unique for any description generated by a grammar, since all such descriptions are complete descriptions. A tree T subsumes a tree T' if and only if all the descriptions which T' satisfies are also satisfied by T ([Rounds and Kasper, 1986]). This definition of the resulting tree is equivalent to the one using the closed world assumption, given above.

A list of strings s is generated by a grammar $G{=}\langle S,\ L,\ A,\ V\rangle$ if and only if s is a sentence for a tree generated by G. A list of strings s is a sentence for a tree T if there is a bijection g from words in s to nonempty terminals in T such that, $g(w)$ is an instance of w and, if $g(u)$ precedes $g(v)$ in T then u precedes v in s. Nonempty terminals are those which are not instances of the empty string. An example of a simple grammar and the formulae generated by it is shown in graphical form in figure 5.

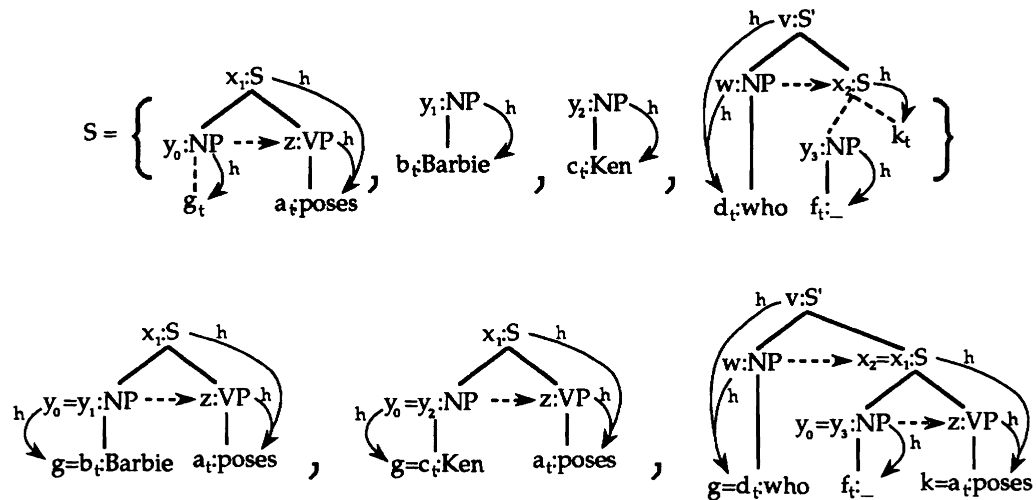


Figure 5: The second row of structure descriptions are those generated by the grammar $G{=}\langle S,\ L,\ A,\ V\rangle$ with S as shown in the first row, $L{=}\{cat,\ head\}$, $A{=}\{$S′,S,NP,VP$\}$, and $x_1,x_2,y_0,y_1,y_2,y_3,z,w,v,a,b,c,d,f,g,k{\in}\ V$.

2.4 Other Formalisms Using Partial Descriptions of Phrase Structure

Many other formalisms can be viewed as combining partial descriptions of phrase structure trees to produce a complete description, but they do not have the properties which will be necessary in the next chapter. These properties are all concerned with the flexibility with which grammatical constraints can be specified. In order to be able to express the large variety of grammatical constraints found in investigations into natural language, it must be possible to state the constraints you want where you want to state them. Other formalisms either do not allow certain constraints to be specified or restrict how these constraints can be grouped into grammar entries. This section will discuss three such formalisms, CFGs, PATR-II, and the system defined in [Rounds and Manaster–Ramer, 1987]. Some other formalisms which can be viewed in this way will be discussed in chapter 3.

One simple formalism which can be viewed as using partial descriptions of phrase structure is Context Free Grammars. Each rule in a CFG specifies a possible tree fragment of depth one. The expansion of a nonterminal in one rule by another rule corresponds to equating a leaf of one fragment with the root of another. In this sense each rule used in a derivation describes a fragment of the final tree. The problem is that the specification of grammatical information in CFGs is very inflexible. Because node labels are atomic, underspecification of these labels is not possible, thus requiring different rules for each way of completely specifying the node label which should be underspecified. The same problem occurs due to the fact that the children of each fragment are completely ordered. Since a CFG grammar entry is limited to being of depth one, any constraint which spans more than one level in the tree can not be expressed in a single grammar entry. Systems of node labels can be devised to encode such constraints, but they suffer from the above limitation on node label specification and lose the perspicuity of the encoded constraint. Finally, because a CFG grammar entry is interpreted as a complete description of the parent–child relationships for its root, all possible combinations of children for the parent must each be specified in a different grammar entry, rather than being able to modularize this specification according to co-occurrence restrictions. These factors prevent the perspicuous representation of many grammatical constraints.

PATR-II ([Shieber, 1986]) is an extension of CFGs which allows node labels to be specified using feature structures, including the ability to specify coreference between features

in node labels. This greatly increases the power of the formalism, but it still suffers from most of the problems discussed above for CFGs. In fact, the only problem it solves is CFGs' inability to underspecify node labels. Through the use of feature passing techniques, this ability in turn helps in the encoding of constraints which span more than one level in the tree, but such feature systems still lose the perspicuity of the encoded constraint. PATR-II still can't underspecify ordering constraints and can't modularize the specification of parent–child relationships according to co-occurrence restrictions.

The only other formalism I will discuss in this section is that described in [Rounds and Manaster–Ramer, 1987], *A Logical Version of Functional Grammar.* This formalism adds to feature structures the ability to specify dominance and linear precedence relations. The resulting logic is used in fixed point formulas to specify grammars. The process of instantiating the type variables in a fixed point formula provides the same kind of structure as CFG derivations, but this structure is not used to enforce ordering constraints. Ordering constraints are enforced using the dominance and linear precedence constraints, which may be unrelated to the variable instantiation structure. This system is less expressive than SUG because it does not have immediate dominance relations or instance unique terminals, but the most important problem arises from the use of fixed point formulas to specify grammars. Just as the instantiation of type variables has the same structure as CFG derivations, the specification of type variables in fixed point formulas has the same restricted domain for specifying grammatical constraints as CFG rules. Any constraint which spans more than one level in the instantiation structure can only be stated with the use of feature passing techniques. The other problems discussed above are avoided in this system because of the extensive use of partial information, including the unrestricted use of disjunction.

Chapter 3

Unifying Insights into Natural Language

Now that we have defined a formalism which allows the flexible use of partial information about phrase structure, it is possible to demonstrate how this approach allows the unification of insights from a variety of investigation into language. Two characteristics of Structure Unification Grammar will be important for this; SUG's use of partial information allows the grammar to say exactly what is known where it is known, and SUG's combination operation permits a complete separation between the phrase structure and the derivation structures. The first characteristic is important for expressing many different kinds of grammatical information, all in a concise perspicuous fashion. The second characteristic is important for expressing within a single framework the insights from both investigations into grammatical constraints on language, and investigations into the processing of language. Both these types of investigations will be discussed.

This chapter will demonstrate how SUG can unify the insights from a variety of investigations into language by discussing an assortment of investigations and showing how insights from them can be captured in SUG. In some cases it will even be possible to show that an insight is better captured in SUG than in its original formalism. The discussion of individual investigations will be preceded by a section which gives examples of how to perspicuously express a variety of grammatical constraints in SUG. These analyses are not specific to any particular investigation, but are taken from the field in general. The subsequent sections discuss particular investigations into language. The investigations discussed are Lexical Functional Grammar, Description Theory, Abney's licensing parser, Tree Adjoining

Grammar, Lexicalized Tree Adjoining Grammar, and Combinatory Categorial Grammar.

3.1 Examples of Expressing Grammatical Constraints

The characteristic of SUG which makes the perspicuous representation of grammatical constraints possible is the flexibility with which information can be specified in the grammar. This flexibility gives SUG a large domain of locality for expressing grammatical constraints, the ability to underspecify information within this domain, and the ability to overlap these domains arbitrarily in a derived structure. The significance of these characteristics will be demonstrated through a series of examples. The first section gives several examples of how SUG's large domain of locality allows the perspicuous representation of lexically specific information within a word's grammar entry. The second section discusses how the underspecification of information can be used to express ambiguities. The third section then discusses how the previous lexicalized grammar entries can be decomposed so as to express generalities in the grammar[1].

3.1.1 Using SUG's Large Domain of Locality

Structure Unification Grammar's large domain of locality for expressing grammatical constraints permits interdependent sets of grammatical constraints to be expressed in single grammar entries. To demonstrate this ability this section will give examples of lexicalized grammar entries. Each of these entries will include a terminal for the lexical item and the fragment of structure necessary to express the grammatical constraints associated with that lexical item. These grammatical constraints are simply what we know about the phrase structure given the presence of the lexical item. This topic will be discussed further in sections 3.5 and 3.6 on Tree Adjoining Grammar and Lexicalized Tree Adjoining Grammar.

The significance of SUG's domain of locality can be demonstrated by contrasting it with that of Context Free Grammars. In CFGs even the enforcement of subcategorization constraints needs to be coordinated between multiple grammar entries. The structure for 'rolls' in the middle of figure 6 shows how several such constraints can be expressed in a single

[1]Throughout this section I will be giving particular analyses, but these analyses are not the point of this section. The objective is to demonstrate that analyses exist which have the properties discussed. Many other analyses are possible within SUG, with varying degrees of naturalness. Disagreements with the analyses given here are not in and of themselves arguments against the claims being made.

SUG grammar entry. The whole projection of the verb is present, the subcategorization for an $\overline{\overline{\text{N}}}$ subject is expressed, and the agreement information is expressed on this subject. To express the interdependence between the lexical item and these constraints in a CFG would require introducing several features of node labels whose sole purpose is to enforce this interdependence across the boundaries of grammar entries. There will be several other examples in this chapter which demonstrate how such node features can be eliminated given SUG's large domain of locality.

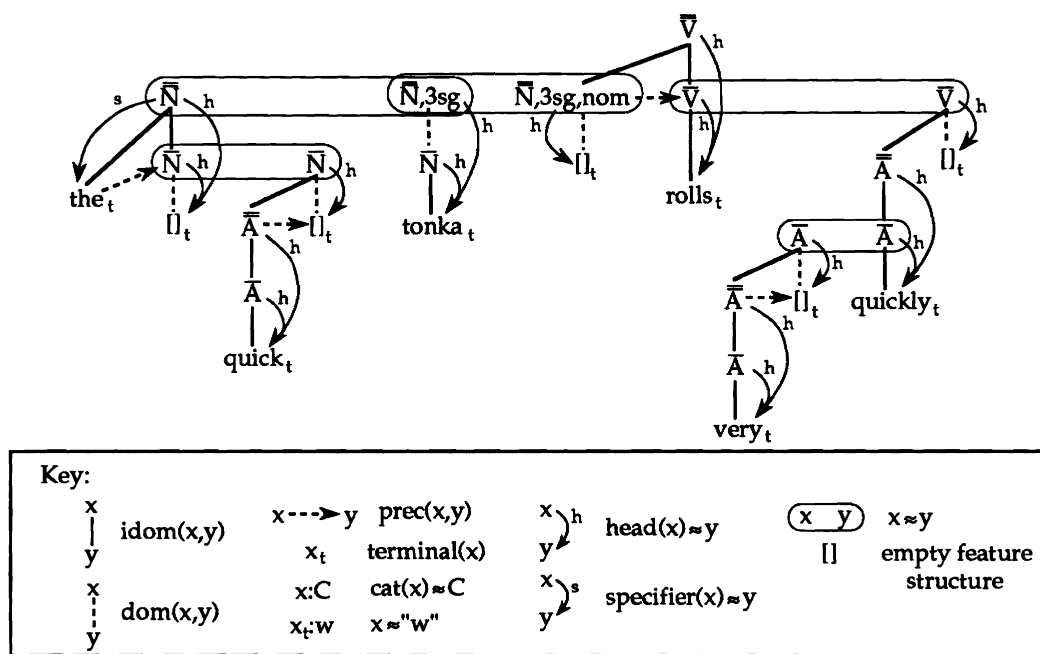


Figure 6: Some example grammar entries used to derive the sentence "the quick tonka rolls very quickly". The key is repeated here for convenience.

The structures for 'quick', 'quickly', and 'very' in figure 6 show how modification information can also be expressed within SUG's domain of locality. This ability eliminates the need to introduce node features which distinguish between the categories adjective and adverb, since the distinction can be expressed within the structure by specifying the category of the modified node. Adjectives are A's which modify N's and adverbs are A's which modify either V's or sometimes other A's. This in turn permits a single entry for 'very' which can modify both adjectives and adverbs. By specifying adjuncts in this way, multiple adjuncts can attach to a single node. The root node of each adjunct structure can equate with the node being modified without interfering with the attachment of the other

adjuncts. This iteration is possible because each adjunct brings with it the link necessary to be attached. In contrast, subcategorized arguments can not iterate because the link for attachment is supplied by the subcategorizing structure, not the argument, and thus only one argument can attach. This technique for attaching adjuncts eliminates the need for Chomsky adjunction[2]. The distinction between adjunct relationships and subcategorization relationships will be discussed further in section 3.4 on Abney's licensing parser.

The structure for 'the' in figure 6 is like the modification structures in that the terminal is not the head of the root, but it cannot iterate because the terminal is the specifier of the root. The link between the $\overline{\overline{\mathrm{N}}}$ and the $\overline{\mathrm{N}}$ is there in case the head of the $\overline{\overline{\mathrm{N}}}$ is not a full $\overline{\overline{\mathrm{N}}}$ by itself, such as is the case for 'tonka' in figure 6. The structure for 'who' in figure 7 has a similar basic configuration. Again in these structures, SUG's ability to express information within the structure associated with a word, rather than just in its category, permits node features to be eliminated. Given this analysis there is no need for the category determiner. In fact the only categories which appear to be needed are the major categories, N, V, A, and P, with their bar levels[3]. This is an indication of how much more expressive a formalism with a large domain of locality, like SUG, is than a formalism like CFGs, in which much of the work in writing a grammar is working out a system of features to enforce constraints across grammar entry boundaries.

Because SUG allows the specification of dominance relations, long distance dependencies can also be expressed in its domain of locality. An example of this is given in the structure for 'who' in figure 7. In this structure node w_1 acts as a trace, since it needs to find an argument position to give it an immediate parent, and it will fill an obligatory argument position by giving the argument node a filled head. The dominance relation restricts w_1 so it must equate to a node within the lower $\overline{\overline{\mathrm{V}}}$, thus enforcing that 'who' must c-command its trace[4], but it also allows w_1 to move arbitrarily far from 'who'. Other constraints on where a trace can equate can be enforced using node features, as will be shown in the discussion

[2]If Chomsky adjunction is desired, then it can be accommodate by splitting $\overline{\mathrm{X}}$ nodes into two $\overline{\mathrm{X}}$ nodes with a dominance link between them. This allows a series of intermediate $\overline{\mathrm{X}}$ nodes to be inserted between them to produce a Chomsky adjunction structure. If nothing is inserted the two $\overline{\mathrm{X}}$ nodes can simply equate, since dominance is recursive, giving the usual unmodified structure. I do not adopt this analysis because I think Chomsky adjunction is an artifact of the inadequacies of CFG, not an insight of that investigation. In terms of the adjunct/argument distinction just discussed, CFGs only have the ability to specify subcategorized arguments.

[3]There are other features, such as tense and agreement, which could be argued to be part of the category of a node. Nevertheless, all nodes can be subcategories of N, V, A, or P and there is no need for any "extracategorial" nodes, such as determiner.

[4]C-command is a relationship often used in Government–Binding Theory. The exact definition is not

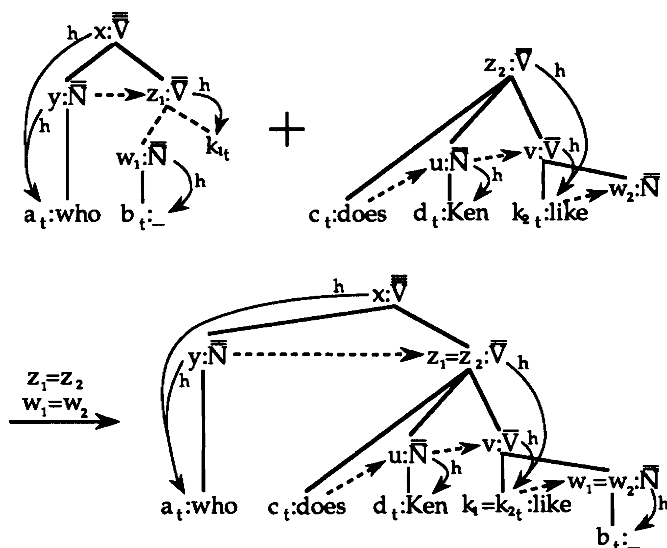


Figure 7: An example of using dominance to express long distance dependencies.

of LFG.

Gerunds are a particularly good test for the domain of locality of a formalism because they act like noun phrases but have the internal structure of verb phrases. Figure 8 gives one possible structure for the gerund 'riding'. This structure includes the usual structure of a $\overline{\overline{V}}$, including the subcategorization for the object of 'ride'. However, the root of the structure is an $\overline{\overline{N}}$[5], thus making it fill $\overline{\overline{N}}$ argument slots.

The two possible structures for 'wants' in figure 9 give another example of the advantages of SUG's domain of locality. The verb 'wants' is followed by an $\overline{\overline{N}}$ and a tenseless $\overline{\overline{V}}$. The $\overline{\overline{N}}$ is semantically the subject of the $\overline{\overline{V}}$, but the $\overline{\overline{N}}$ gets its Case[6] from 'wants'. This leads to two possible structures for 'wants', one which follows the semantic structure and one which follows the Case structure, as shown in the first and second structures in figure 9,

always agreed upon, but it always involves there existing a node which is a short distance above the c-commanding node and an arbitrary distance above the c-commanded node. In this case this node is the top $\overline{\overline{V}}$.

[5]The head of the $\overline{\overline{N}}$ root is shown as being 'riding', which is also the head of the $\overline{\overline{V}}$. This seems to violate the linguistic notion of head, but as is discussed in section 3.1.3, the head of the $\overline{\overline{N}}$ is actually the '–ing' ending.

[6]In Government-Binding Theory, Case is a formal notion closely related to case. All overt $\overline{\overline{N}}$'s must receive Case, even if their case is not overtly marked. 'Wants' is an exceptional Case marking verb because it assigns Case to the semantic subject of its object, rather than having the subordinate verb assign Case, as is true for 'said'.

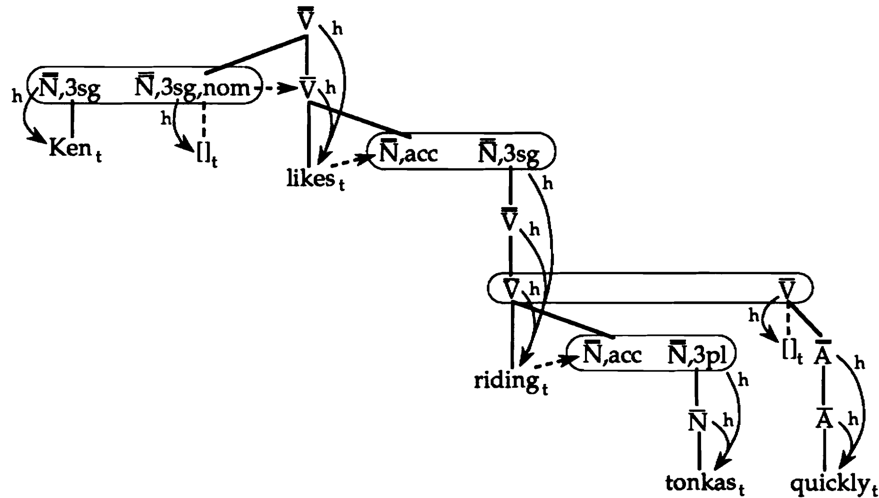


Figure 8: One possible grammar entry for the gerund ‘riding’ used to derive the sentence “Ken likes riding tonkas quickly”.

respectively[7]. In either structure the relationship not expressed in the structure can be expressed over the nodes in the structure, and the case of the $\overline{\overline{\mathrm{N}}}$ can be expressed. I will adopt the second of these structures because semantic information will have to be expressed anyway[8], so it seems unnecessary to have the syntactic structure mimic the semantic structure. Also, Case seems like an inherently syntactic phenomena, so it is not clear what role it would play if not to determine the syntactic structure. Thus it is natural to assume that immediate dominance links act analogously to Case assignment, only extended to all the categories. Under this interpretation the adjunct structures in the previous structures can be interpreted as saying that adjuncts assign themselves Case. This interpretation of Case is similar to Abney’s notion of licensing, as will be discussed in section 3.4.

[7]There are other possible analyses. One common analysis is to express the subcategorization for the subject with the infinitival verb, in the same way as would be done for tensed verbs, except the subject would be marked as needing Case. Given this, the structure for ‘wants’ would still need to mention the subject in its structure in order to say that it gives the subject Case. In accord with the idea that the grammar entry should say everything known, the subcategorization and subjecthood relationship would also be expressed in the structure for ‘wants’. Given this, it is not clear why the subcategorization for the subject should also be in the structure for the infinitival verb. For this reason I have not included this analysis in the example, but that is not to say it could not be done.

[8]The need to express semantic information separately from phrase structure relations will be argued for in section 3.2 on Lexical Functional Grammar.

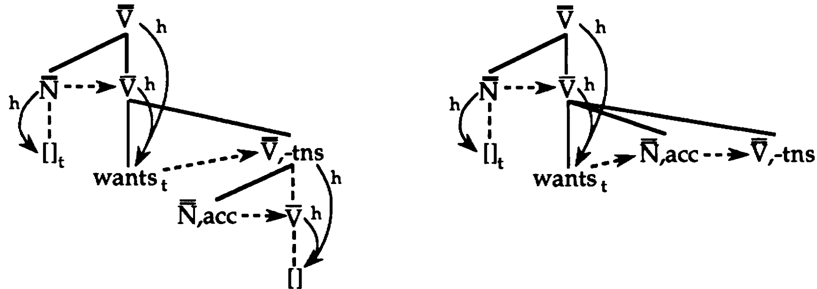


Figure 9: Two possible grammar entries for the exceptional case marking verb 'wants'.

3.1.2 Trading Ambiguity for Underspecification

SUG not only provides the domain of locality necessary to state what constraints are known where they are known, it also allows you not to say what you don't know. This is a natural consequence of using partial specifications. By underspecifying information, what would otherwise be an ambiguity between multiple grammar entries can be expressed in a single grammar entry. This section will give a few examples of this ability.

Figure 10 shows how underspecification of node labels can be used to express ambiguities. Because feature structures are being used to label nodes, it is possible to underspecify these labels, and thus express ambiguity between multiple labels. To do this, however, it is necessary to use a feature decomposition of node labels which allows the desired ambiguities to be expressed. In the examples in figure 10 I use a feature decomposition of the major categories which differs from the Chomskian feature decomposition. I represent N as [−S,−M], V as [+S,−M], A as [−S,+M], and P as [+S,+M][9]. This allows one structure for 'know' which allows for either an $\overline{\overline{N}}$ or a $\overline{\overline{V}}$ object, which would not be possible with the Chomskian feature decomposition. The second structure in figure 10 allows 'always' to attach to either a $\overline{V}$ or a $\overline{P}$.

Another kind of ambiguity was expressed in each possibility for the structure for 'wants' given in figure 9. Both structures express the fact that the objects of 'wants' are both optional. They are optional because there are no underspecified terminals associated with them. Remember that in order for a description to be complete, all the terminals in the description need to have their words specified. By giving a node an underspecified terminal as its head, that node must equate with a node which has a word as its head, thus "filling"

[9]Intuitively, the S feature stands for "usually subcategorizes for something" and the M feature for "usually modifies something". Of course this interpretation of these features has no causal role in the system.

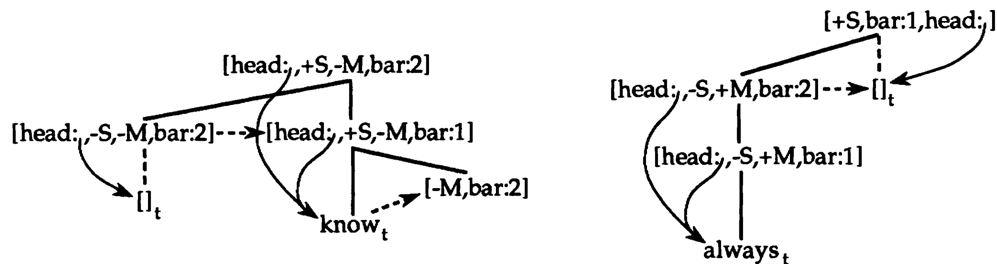


Figure 10: Examples of using feature structures to underspecify node labels.

the argument. This technique is used to make the subject of 'wants' obligatory. Because the heads of the two objects of 'wants' are either not mentioned or not specified as terminals, they do not have to be equated with for the structure to be complete. Thus the objects are optional.

3.1.3 Capturing Generalities

All the examples of grammar entries in the above sections are lexicalized. Using different grammar entries for each word fails to express the generalities in the grammar. For example, the structure given for 'rolls' in figure 6 has a lot in common with the structure given for 'likes', as will any tensed verb. To express this generality it is necessary to split the information in these structures into the part which is present simply because the terminal is a tensed verb, and the part which is specific to this verb. In SUG this can be done, because the combination operation SUG uses allows two different grammar entries to have multiple nodes in common in the derived structure. This section will look at several of the structures given in the previous section and show how they can be constructed from a more modular set of structures which express generalities in the grammar.

Figure 11 shows how the structures for 'rolls' and 'likes' given above can be split up to express the significance of tense in the grammar. The root of each verb determines what objects are subcategorized for, and the tense, which is manifested as a '–s' ending on the verbs, determines the subcategorization for the subject. When the verb root's structure is combined with tense's structure by equating the $\overline{\overline{V}}$'s, the result is the structure given above for the tensed verb. Note that the verb root and tense structures have the entire projection of the verb in common after they are combined. Because such overlapping is allowed, structures can be split according to the interdependence of information, rather than according to the topology of the structure.

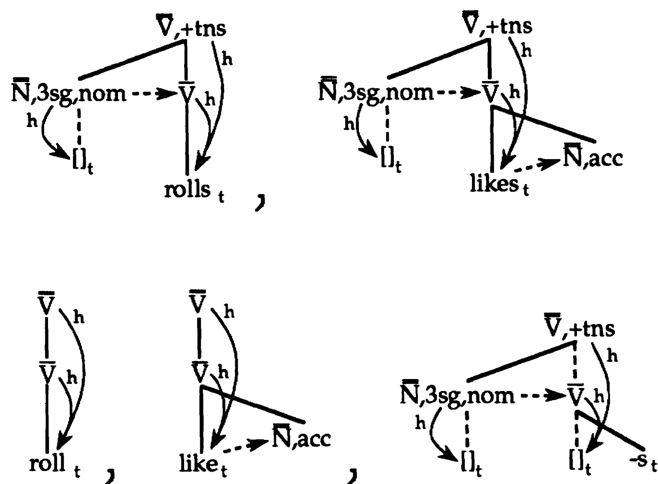


Figure 11: The two structures given in the first line can be split by morpheme and expressed as the three structures given in the second line. The later decomposition expresses the significance of the tense suffix '–s' in the former.

A similar split to that just discussed can be used to express the significance of the '–ing' ending on gerunds. Figure 12 shows how the structure for 'riding', taken from figure 8, can be split into a structure for 'ride' and a structure for '–ing'. As in the above paragraph, the verb root specifies the projection of the verb and what objects are subcategorized for. The '–ing' ending specifies the $\overline{\overline{N}}$ root and its relationship to the $\overline{\overline{V}}$. Again, the flexibility of the combination operation is necessary to allow this split. Also, as is the case in all the structures given here, the information provided in the structure for 'ride' is exactly what is known about the structure given the presence of this morpheme, and the structure for '–ing' specifies exactly what is known about the structure given the presence of the morpheme '–ing'.

Splitting lexicalized grammar entries into a component for each morpheme can also be used to express the significance of a word's root in the grammar. This is demonstrated in figure 13. 'Quick' can be the root of either an adjective or an adverb, although in the former case the affix is not phonetically realized. The structure for this root specifies its projection, which presumably includes semantic information. The '–ly' suffix makes 'quick' an adverb by specifying that the $\overline{\overline{A}}$ modifies a $\overline{V}$. The other structure shown allows 'quick' to become an adjective by specifying that the $\overline{\overline{A}}$ modifies an $\overline{N}$, although the effects of this structure are not phonetically realized in the resulting terminals.

Structures like those given above obviously do not capture all the generalities which

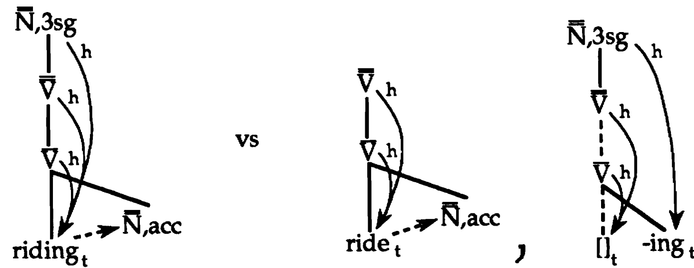


Figure 12: The structure given on the left can be split by morpheme and expressed as the two structures given on the right. The later decomposition expresses the significance of the verb root and the suffix '–ing' in the former.

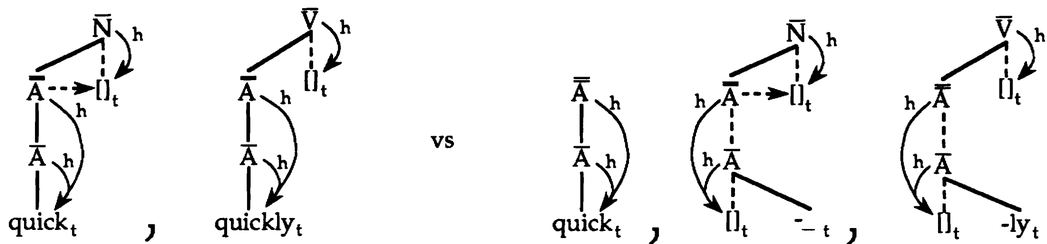


Figure 13: The two structures given on the left can be split by morpheme and expressed as the three structures given on the right. The later decomposition expresses the significance of the root 'quick' in the former.

exist in the grammar. However, too fine grained splitting of structures would force the introduction of special features to enforce constraints between different grammar entries. This is precisely what was being avoided in the discussion of SUG's large domain of locality. At the moment it is not clear what degree of modularity is appropriate for SUG's level of representation. It could be that in order to fully express linguistic generalities without nullifying the advantages of SUG's large domain of locality, another level of representation would need to be introduced. Such a level could be analogous to meta–rules in GPSG[10].

3.2 Lexical Functional Grammar

Capturing Lexical Functional Grammar's expressive abilities within SUG is of interest because it demonstrates how an SUG grammar can enforce a broad set linguistic constraints. Of the many linguistic formalisms, Lexical Functional Grammar (LFG) is of particular interest because of its explicit representation of semantic information in a structure distinct from the phrase structure. This method of encoding semantic information in a grammar works well in SUG and fits well with the approach that a grammar should explicitly say what is known where it is known. LFG is also of interest here because the complexity and directness of the formalism's representations exceeds that of the other formalisms discussed in this paper.

To demonstrate that Lexical Functional Grammar's expressive abilities can be captured within SUG I will show that any grammar in the version of LFG to be discussed can be translated into an equivalent SUG grammar[11]. The fact that SUG is sufficiently powerful to do this at all is interesting, but also interesting is the way various constraints from an LFG grammar are expressed in an SUG grammar. The constraints which will be of particular interest are the explicit encoding of predicate–argument structure within the grammar and the handling of long distance dependencies.

After specifying the precise version of LFG which I will be using, this section defines the translation from such an LFG grammar to an SUG grammar. Following the definition of the translation, the expression of linguistic constraints in the two formalisms will be compared.

[10]Tilman Becker is investigating such a meta–level representation for Tree Adjoining Grammar.

[11]Since LFG is known to be undecidable, this fact implies that SUG is undecidable as well. However, if the feature structures of LFG grammars are limited so they can not generate arbitrarily large feature structures, then LFG is decidable, and so is the translation of LFG in SUG. This issue will be discussed further in section 3.5 on Tree Adjoining Grammars.

3.2.1 The Version of LFG

The version of LFG which I will use here is a subset of that given in [Kaplan and Bresnan, 1982]. This version was chosen over more recent versions which involve functional uncertainty because functional uncertainty is less perspicuously represented in SUG than the system of bounded dominance metavariables and bounding nodes used in [Kaplan and Bresnan, 1982]. It is not clear that functional uncertainty can be completely simulated in SUG at all. The only other serious shortcoming of the version of LFG used here is the lack of set valued features in f-structures. To prevent any confusion with more complete versions, the version presented below will be called LFG′.

3.2.1.1 Constituent Structure

In [Kaplan and Bresnan, 1982], constituent structure (c-structure) is determined by context free rules with a few additions. The categories on the right side of a rule may be placed in parenthesis, followed by a star, or a set of categories can be placed in braces and followed by a star. The parenthesis denote that that constituent is optional. The star denotes that zero or more instances of the category may be present in that position. The braces followed by a star says that zero or more instances of any of the categories in the braces can be present in that position, in any order. The f-structure equation associated with such a repeated category is applied to each instance of that category, not to the collection of them. All these features are allowed in LFG′.

3.2.1.2 Functional Structure

Most of the work in LFG is done in the functional structure (f-structure). F-structures are represented in feature structures. The features and values of these feature structures are specified with equations associated with each c-structure node. These equations can be expressed using immediate dominance metavariables or bounded dominance metavariables. The features and values may also be constrained with other statements annotating each c-structure node.

The Feature Structures

The feature structures used in [Kaplan and Bresnan, 1982] have a few special characteristics. Although a given grammar has a finite number of labels, there are an infinite number

of possible atoms. This is because each instance (token) of a semantic form is unique. Other than semantic forms, a given grammar has a finite number of atoms. In LFG′ the use of semantic forms is restricted to occurring as values to the feature PRED. In addition to regular atoms, semantic forms, and feature structures, [Kaplan and Bresnan, 1982] also allows features to have set values. Since there are no set values in SUG, LFG′ does not allow the use of set values. As will be discussed in section 3.2.5, this slightly restricts the coverage of this version of LFG, as compared to that of [Kaplan and Bresnan, 1982].

Local Feature Value Statements

One way to specify information about the f-structure is by using the immediate dominance metavariables ↓ and ↑. The ↓ is instantiated with the ↓-variable of the node whose category the equation annotates. The ↑ is instantiated with the ↓-variable of the node whose category is on the left side of the rule. These variables are given the values of the ↓ f-structures of the nodes, henceforth simply called the f-structures of the nodes. Thus the equations discussed here state information about these f-structures. For example, (1)[12] says that the value of the SUBJ feature of the f-structure for the S node is the f-structure for the NP node, and the f-structure of the S node is the same as that for the VP node. Such equations can also be stated on lexical entries, as shown in (2).

(1) (K's 21) $S \longrightarrow \begin{matrix} NP \\ (\uparrow \text{SUBJ}) = \downarrow \end{matrix} \quad \begin{matrix} VP \\ \uparrow = \uparrow \end{matrix}$

(2) (K's 22) *handed:* V, $\begin{matrix} (\uparrow \text{TENSE}) = \text{PAST} \\ (\uparrow \text{PRED}) = \text{`HAND}\langle(\uparrow \text{SUBJ})(\uparrow \text{OBJ2})(\uparrow \text{OBJ})\rangle\text{'} \end{matrix}$

In [Kaplan and Bresnan, 1982] immediate dominance metavariables are used in two ways, to specify two values as being equal, and to specify a label as being the same as a value. The above examples are of the first type. (3) shows the second use. Here the feature

(3) (K's 43) $VP \longrightarrow V \left(\begin{matrix} NP \\ (\uparrow \text{OBJ}) = \downarrow \end{matrix}\right) \left(\begin{matrix} NP \\ (\uparrow \text{OBJ2}) = \downarrow \end{matrix}\right) \begin{matrix} PP^* \\ (\uparrow (\downarrow \text{PCASE})) = \downarrow \end{matrix}$

which the PP's f-structure is a value of, is labeled by the symbol which is the value of the PP's f-structure's PCASE feature. Both these uses are allowed in LFG′, but the second can

[12]Most of the examples in this section are taken from [Kaplan and Bresnan, 1982]. The numbers in [Kaplan and Bresnan, 1982] for the examples will be given next to the numbers used here.

not be used to specify a label to be the value of a PRED feature because PRED's value is a semantic form.

In addition to being able to specify a specific value for a feature, it is also possible to use disjunction to specify that a feature has one of several possible values. This is also allowed in LFG′.

Feature Value Constraints

In addition to being able to state the value of a feature, it is also possible to constrain it in various ways[13]. One way is demonstrated in example (4). Here the CASE feature

(4) he: N, $(\uparrow \text{NUM}) = \text{SG}$
$(\uparrow \text{CASE}) =_c \text{NOM}$
$(\uparrow \text{PRED}) = \text{'PRO'}$

of the f-structure for the N node is constrained to be NOM. This means that the CASE feature must be stated as having the value NOM by some other rule; it does not actually state the value of this feature. The difference between this and a normal equation is that the f-structure is also ruled out if no value is specified for the feature.

Another way to constrain the value of a feature is simple to specify that there must be a value, without specifying what that value must be. This is called an existential constraint and is demonstrated in (5) by the statement (↑ TENSE).

$$\text{(5) (K's 71) S} \longrightarrow \begin{array}{cc} \text{NP} & \text{VP} \\ (\uparrow \text{SUBJ}) = \downarrow & \uparrow = \downarrow \\ (\downarrow \text{CASE}) = \text{NOM} & (\uparrow \text{TENSE}) \end{array}$$

Either of these constraint statements can also be used with a negation symbol to constrain what a feature's value can't be. For example, (6) says that if *to* is present then the f-structure of the S node can't be specified for the TENSE feature. In LFG′, the value of a feature which is negatively constrained can only range over atoms[14].

$$\text{(6) (K's 73) VP}' \longrightarrow \begin{pmatrix} \textit{to} \\ \neg\,(\uparrow \text{TENSE}) \end{pmatrix} \begin{array}{c} \text{VP} \\ \uparrow = \downarrow \end{array}$$

[13]None of the methods of constraining feature values can be used for the PRED feature, since its value is a semantic form. The importance of this restriction will be demonstrated in section 3.2.2.2.

[14]It is not clear that this restriction is necessary, but it makes the simulation of this mechanism easier.

Nonlocal Feature Value Statements

In addition to the immediate dominance metavariables, an f-structure statement may include the bounded dominance metavariables $\Uparrow$ and $\Downarrow$. Each $\Uparrow$ is instantiated with the same $\Downarrow$-variable as some $\Downarrow$. This relationship is called constituent control. The $\Downarrow$ is called the constituent controller and the $\Uparrow$ is called the constituent controllee. Constituent control can be constrained by adding subscripts to the arrows. One bounded dominance metavariable can control another only if they have the same subscripts. It is also necessary for the $\Uparrow$ to be within a control domain for the $\Downarrow$. A control domain is a portion of the c-structure dominated by the control domain's domain root. The domain root of a control domain for a $\Downarrow$ can be restricted by adding a superscript which specifies the category of the domain root. The domain root is also required to be a child of the parent node in the rule in which the $\Downarrow$ is specified. Thus, in (7), $\Downarrow^{S}_{NP}$ must control a $\Uparrow_{NP}$, such as that in (8), and this $\Uparrow_{NP}$ must be in an equation annotating a node in the control domain rooted by the S node.

(7) (K's 141) $S' \longrightarrow \begin{array}{cc} NP & S \\ (\uparrow Q) = \Downarrow^{NP}_{[+wh]} & \uparrow = \downarrow \\ (\uparrow FOCUS) = \downarrow & \\ \downarrow = \Downarrow^{S}_{NP} & \end{array}$

(8) (K's 135) $NP \longrightarrow \begin{array}{c} e \\ \uparrow = \Uparrow_{NP} \end{array}$

Control domains are limited by specifying bounding nodes. These are represented by putting boxes around the bounding nodes in a c-structure rule, as shown in the more complete version of (7) given in (9). The complete definition of a control domain is given

(9) (K's 150) $S' \longrightarrow \begin{array}{cc} NP & \boxed{S} \\ (\uparrow Q) = \Downarrow^{NP}_{[+wh]} & \uparrow = \downarrow \\ (\uparrow FOCUS) = \downarrow & \\ \downarrow = \Downarrow^{S}_{NP} & \end{array}$

in the bounding convention ([Kaplan and Bresnan, 1982] p245):

> A node M belongs to a control domain with root R if and only if R dominates M and there are no bounding nodes on the path from M up to but not including R.

Set Valued Feature Structures

As mentioned above, [Kaplan and Bresnan, 1982] uses set valued feature structures. Since these are not available in SUG, LFG′ must be defined differently than the version in [Kaplan and Bresnan, 1982] for certain phenomena. [Kaplan and Bresnan, 1982] analyzes adjuncts and coordination using sets. In LFG′, adjuncts are analyzed by adding what would be included in a set directly to the feature structure the set would be in. This may require changing some feature labels to prevent unwanted feature clashes. LFG′ does not include any provisions for handling coordination. This is because in SUG coordination appears to be best treated at the level of processing. This view will be discussed below in section 3.7 on Combinatory Categorial Grammar.

3.2.1.3 Global Constraints

There are several well-formedness constraints on f-structures. The requirements given in [Kaplan and Bresnan, 1982] are functional uniqueness, completeness, coherence, and proper instantiation. In addition LFG′ requires unique association. Functional uniqueness says that in any f-structure a particular feature can have at most one value. This is enforced by the fact that f-structures are represented as feature structures. The rest of the constraints are discussed below.

Completeness ensures that all the features needed by all the f-structures' PRED features are present. An f-structure is complete if it and all the f-structures in it are locally complete. An f-structure is locally complete if it contains values for all the governable features which its predicate governs. A feature is governable if any semantic form in the grammar subcategorizes for it. An f-structure's predicate governs a feature if the f-structures PRED feature's semantic form mentions the feature.

Coherence is the complement of completeness. An f-structure is coherent if it and all the f-structures in it are locally coherent. An f-structure is locally coherent if all the governable features it contains are governed by the f-structure's predicate. In addition to this, the definition of coherence is extended to include "topicalized" categories. For an f-structure to be locally coherent, the features TOPIC and FOCUS must have their values identified with those of features which are subcategorized for. [Kaplan and Bresnan, 1982] also allows the values of TOPIC and FOCUS to be anaphorically bound in stead of identified, but since anaphora is not being considered here, LFG′ does not allow for this possibility.

Proper instantiation restricts the instantiation of bounded dominance metavariables

beyond requiring that a controllee be in a control domain of the controller it is bound to. Several conditions must be satisfied for an f-structure to be properly instantiated. All domain roots must be distinct. There must be at least one control domain for each controller ($\Downarrow$). For a given controller, each of its control domains must have exactly one controllee ($\Uparrow$). Each controllee must have exactly one controller. [Kaplan and Bresnan, 1982] also has requirements that the binding relationships be "nearly nested" and that all control domains have a lexical signature. These are not part of LFG′. The first is probably a processing constraint and the second is best dealt with as part of a theory of headedness.

Unique association is a restriction on allowable LFG′ grammars. It requires that for any grammar there can be defined a partial mapping from metavariables to categories in c-structure rules such that, for every f-structure F generated by the grammar, this mapping determines a total function from the f-structures included in F to the c-structure nodes of F's associated c-structure. In other words, for every f-structure included in a generated f-structure, there is always a unique c-structure node which is associated with it via a mapping defined on the grammar. In all the examples in this section, the mapping can be defined by associating each category which is annotated with an equation assigning a value to PRED, with the metavariable having this feature predicated of it. Since all the f-structures in the examples will eventually have one and only one value for PRED, this mapping fulfills the requirements for unique association. The importance of this restriction for the simulation of LFG′ in SUG will be discussed in section 3.2.2.2.

3.2.1.4 Anaphora

[Kaplan and Bresnan, 1982] includes a theory of anaphoric binding as part of LFG. Since SUG provides no method for coindexing things, other than having them share some feature structures, LFG′ will not include any theory about anaphoric binding. In SUG anaphoric binding is considered a postsyntactic phenomena.

3.2.2 Expressing LFG′ in SUG

In this section the mapping from an LFG′ grammar to an equivalent SUG grammar is given. The SUG grammars are equivalent to their associated LFG′ grammars in the sense that they define the same sets of sentences, and portions of the SUG structures produced for each sentence can be identified which are the same as the f-structures produced by the LFG′ grammar for those sentences. In particular, the f-structure is a subset of the

SUG structure's root's feature structure. In addition, the c-structures for a sentence are approximately a subset of the nodes and relations in the SUG structure, each labeled with one of the values in their feature structure. The exact definitions of these correspondences will be given below. The discussion below will parallel that in the previous section. At the end of each component of the mapping, the significance of that component will be briefly discussed.

3.2.2.1 Simulating Constituent Structure Rules

An SUG grammar can simulate c-structure rules by augmenting the simulation of context free rules. Context free rules can be simulated as shown in figure 14. The structural relationships between nodes are the same as the derivation structure for this rule, with the addition of a few extra terminals. Every nonterminal here has a *cat* feature which specifies the category of that node, and a *uid* (Unique IDentification) feature which is used to ensure that all possible equations simulate application of CFG rules. The *uid* features of the roots of these treelets have an empty string terminal as their value. Since all such roots have this feature and terminals are instance unique, two roots can never equate. In addition, the nonterminal leaves of a treelet all have distinct values for the feature *position*, so they can never equate to each other. Thus the only allowable equations are between leaves and roots. Such an equation simulates the expansion of the leaf by the rule for the root's structure.

To guarantee that the leaves of a rule's structure do equate with the root of another rule's structure, the nonterminal leaves of these treelets have underspecified terminals as the values of their *uid* features. In order for the structure to be complete, these underspecified terminals must equate with fully specified terminals. The only way this can happen is if the leaves which have the terminals as their *uid* values equate with roots, which have specified terminals as their *uid* values. Once a leaf is equated with a root it can't participate in any more equations due to its *uid* feature value. Note that the existence of the empty string terminal in this structure has no linguistic significance. The notation of SUG could easily be changed to eliminate the need for this terminal without changing its power or basic character.

Given this framework, the use of parenthesis to designate an optional constituent can be easily simulated, as shown in figure 15. The only difference between the optional and obligatory arguments is that optional arguments do not have their *uid* feature referring to an underspecified terminal. This means that the optional argument leaves do not need to

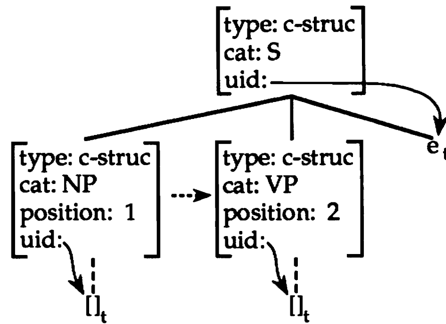


Figure 14: The SUG structure which simulates the context free rule S ⟶ NP VP.

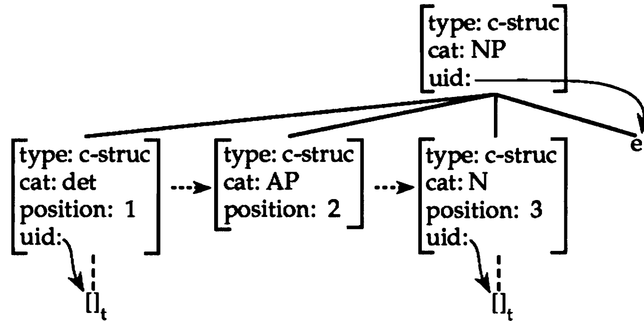


Figure 15: Simulating the optional argument in NP ⟶ DET (AP) N.

find a root to equate with, but there is still nothing preventing such an equation from taking place. Thus the expansion of this constituent is optional, as desired.

Simulating the two additions involving repeated constituents is a little harder. This requires the addition of a node which is not in the c-structure, and interaction with other rules. The effects of following a category with a star is simulated as shown in figure 16. The position of the starred category is recorded by adding a special node in that position. This node has the value **-node* for the feature *type* so as to distinguish it from the c-structure nodes. The features *rule#* and *position* make this node distinct from other uses of star. The possibility of having an arbitrary number of constituents in this position is handled by allowing each child to provide its own immediate dominance link to this special node. Thus every rule which expands a category of the type which is starred needs an additional structure for attaching in this position. In figure 16 one of these additional structures is given for the rule PP ⟶ P NP. The root of this structure can only equate with the particular star node for NP ⟶ DET N PP* because of the values given to its *type*, *rule#*,

and *position* features. In this way, any number of any PP expansion structures can be introduced in this position. This method can be generalized to the cases where the star follows a set of categories by introducing the additional structures for the rules expanding any categories in the set. This technique for encoding stared constituents is the same as the usual way of expressing them in CFG, except Chomsky adjunction is not necessary.

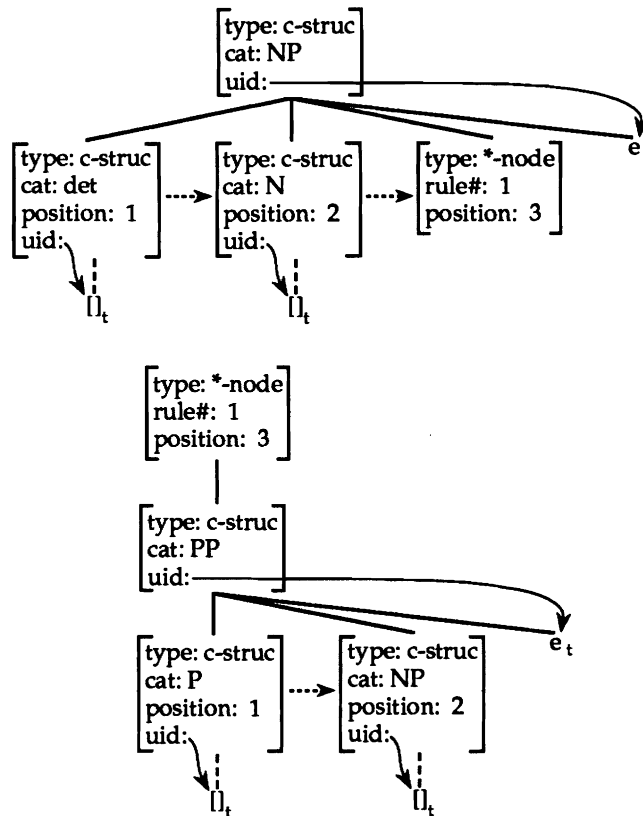


Figure 16: The structure for the rule NP ⟶ DET N PP* and the additional structure this rule needs for the rule PP ⟶ P NP.

As this last construction indicates, the structure constructed for a sentence by an SUG grammar may not be exactly the same as the c-structure constructed for that sentence by the equivalent LFG′ grammar. However, there is a straightforward mapping between the two. Given a structure produced for a sentence by an SUG grammar, the c-structure which would be produced for this sentence by the equivalent LFG′ grammar is the tree formed by the immediate dominance and linear precedence relations between the nodes whose *type* feature has the value *c-struc*, plus immediate dominance relations between each child of a

node whose *type* feature has the value **-node*, and this later node's parent. Each c-structure node has the label given in the associated SUG structure node's *cat* feature. This mapping will not change even though other types of nodes will be added to the SUG structure in the following section.

The one remaining problem in simulating c-structure rules is how lexical entries are used to close off the expansion of a category. As demonstrated in figure 17, lexical entries are treated as simple c-structure rules. Since the *uid* feature of the root is a terminal with its word specified, equating it to a node will close off the simulated expansion of this later node.

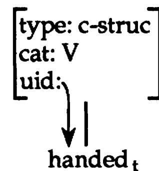


Figure 17: The structure for the lexical entry "handed: V".

3.2.2.2 Simulating Functional Structure Annotations

F-structure is represented in the SUG grammars by embedding it in the feature structures of the nodes described in the previous section. In this way the instantiation of immediate dominance metavariables is a direct consequence of the unification of the node's feature structures. However, this does not allow the direct expression of many of the mechanism LFG′ uses for specifying information about f-structures. The simulation of these mechanisms often require enumerating many cases or making use of the constraints on SUG structures using additional nodes. The instantiation of bounded dominance metavariables can also be simulated straightforwardly by embedding their f-structures in the feature structures of nodes. The difference between this process and simulating immediate dominance metavariable instantiation is that simulating bounded dominance metavariable instantiation makes use of the ability to only specify dominance relations, thus allowing the metavariables to control another metavariable which is an arbitrary distance from the c-structure nodes where it is attached. Each of these constructions will be discussed in detail below.

Representing LFG′ Feature Structures

The feature structures used in SUG are very similar to those used in LFG′, but one difference needs to be compensated for. LFG′ grammars have an infinite number of atoms, since semantic forms are instance unique. Thus it is not adequate to simple specify semantic forms as atoms in SUG feature structures. However, the effect of instance unique atoms can be achieved by representing each semantic form as a feature structure containing an atom which specifies the semantic form, and a terminal, since terminals are instance unique. With this minor addition, LFG′ feature structures can be translated directly into SUG feature structures.

Simulating Local Feature Value Statements

Statements which specify the values of attributes in ↓ f-structures can be simulated by simply recording this information in the feature structures which represent these local variables. These feature structures are the values of the feature *f-struc* in each c-structure node. Figure 18 shows how (10)[15] is represented in this way. The fact that the value

$$(10)\ S \longrightarrow \begin{array}{cc} NP & VP \\ (\uparrow \text{SUBJ}) = \downarrow & \uparrow = \downarrow \\ (\downarrow \text{CASE}) = \text{NOM} & \end{array}$$

of the SUBJ feature in the f-structure of the S is the f-structure of the NP, is stated by coreferencing these two values. Now when a node is equated with the NP node, the former node's f-structure will be unified with the value of the *subj* feature for the S node. Lexical entries are handled similarly, as shown in figure 19[16]. For structures which include a node with *type: *-node,* the *f-struc* value of this node is coreferenced with that of the parent. The equations annotating the starred symbol are then represented on the additional structures for the rules expanding this starred symbol. This is discussed further below and is demonstrated in figure 20.

Given this representation for the f-structure of a given c-structure node, the f-structure of a sentence is the value of the *f-struc* feature in the root of the sentence's complete SUG structure, converted as discussed in the previous section. Later it will actually be necessary to change this definition slightly and eliminate a few features from this feature structure

[15] My example, but derived from (71) in [Kaplan and Bresnan, 1982].

[16] As in [Kaplan and Bresnan, 1982], the metavariables within the semantic form are left unanalyzed, since their instantiation plays no role in the acceptability of a structure.

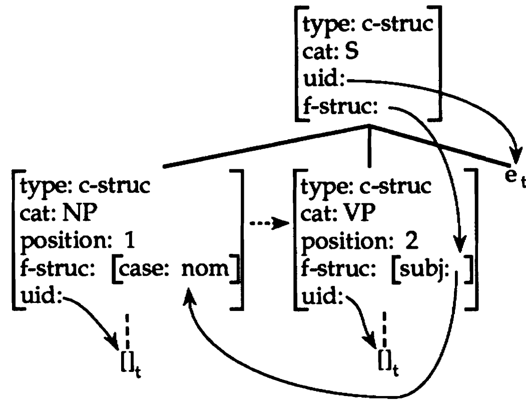


Figure 18: A partial simulation of (10) by adding the *f-struc* feature to figure 14.

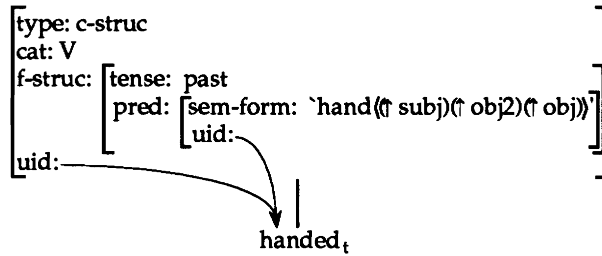


Figure 19: The simulation of (2), repeated below, by adding the *f-struc* feature to figure 17.

(2) (K's 22) *handed:* V, $(\uparrow \text{TENSE}) = \text{PAST}$
$(\uparrow \text{PRED}) = \text{`HAND}\langle(\uparrow \text{SUBJ})(\uparrow \text{OBJ2})(\uparrow \text{OBJ})\rangle\text{'}$

and the feature structures it includes. However it will remain a simple matter to extract the f-structure from a SUG structure for a sentence.

Since there is no way to underspecify or coreference feature labels in SUG feature structures, the use of immediate dominance metavariables to specify equality of a label with a value can not be expressed directly. However, since any LFG′ grammar has only a finite number of labels, it is always possible to enumerate all the possible labels and enforce this equality in every enumeration. Since SUG derivation is a nondeterministic process, it is sufficient to simply list all these possibilities in the grammar. An example of a schema for these entries for (11) and (12) is given in figure 20. Note that the *case* feature's value is

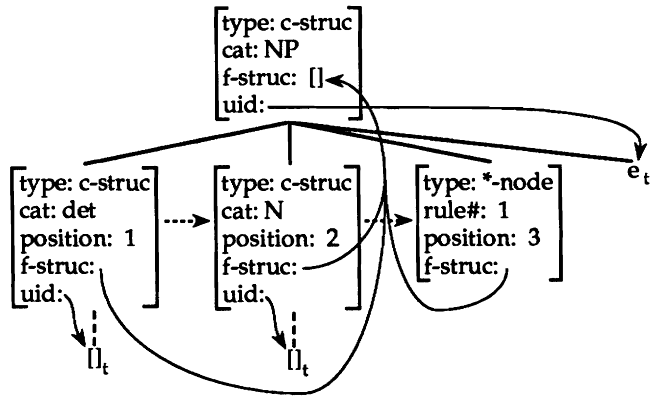
$$\text{(11) NP} \longrightarrow \quad \underset{\uparrow = \downarrow}{\text{DET}} \quad \underset{\uparrow = \downarrow}{\text{N}} \quad \underset{(\uparrow\ (\downarrow\ \text{CASE})) = \downarrow}{\text{PP*}}$$

$$\text{(12) PP} \longrightarrow \quad \underset{\uparrow = \downarrow}{\text{P}} \quad \underset{(\uparrow\ \text{OBJ}) = \downarrow}{\text{NP}}$$

written as "case:$_c$ α". This is not actually an expression within SUG feature structures; it denotes that the value of *case* is constrained to be α, but this is not yet known to be true. The expression of such constraints within SUG will be discussed in the next section.

The use of disjunction to specify feature values is also handled by taking advantage of SUG's nondeterministic derivation process. All that need be done is specify a different structure for each disjunct. If there is more than one disjunction in a rule, then there must be a different structure for each possible combination of values. This in effect pushes all disjunction down into the grammar, which is disjunctive by nature. It would be possible to add to SUG the ability to specify arbitrary disjunction, as is done with feature structures in [Rounds and Kasper, 1986]. This would not change the formal power of SUG, but most linguistic applications don't seem to need it. Whenever possible disjunction should be expressed as the underspecification of feature values.

As is hopefully now clear, the local feature value statements for the f-structures of LFG are easily expressed in SUG as part of SUG's node labels. This technique carries over directly to methods of specifying syntactic structure other than that used in LFG. The predicate–argument structure specified in LFG as annotations on CFG rules can be specified in any SUG grammar as information embedded in the feature structure labels of



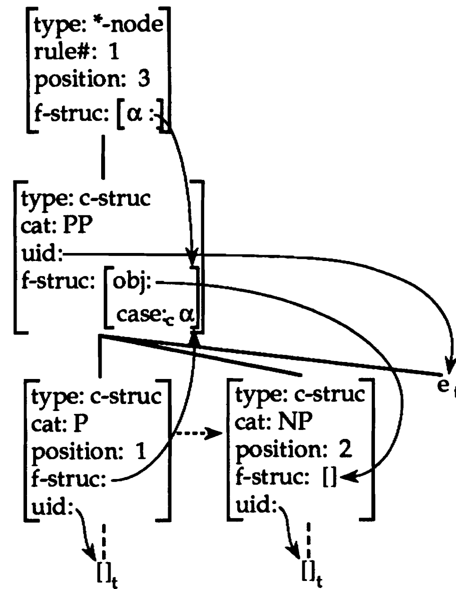


Figure 20: Schema for the structure for (11) and the additional structure this rule needs for (12). This is figure 16 with the *f-struc* feature added.

nodes. This is facilitated in SUG because SUG's domain of locality is sufficiently large to specify these predicate–argument relations without the feature passing techniques necessary in LFG.

Enforcing Feature Value Constraints

The ability to constrain the value of a feature is very difficult to simulate in SUG. It requires the introduction of a new type of node and the use of the completeness requirements for SUG structures. The basic idea is that a constraint equation introduces a new node which has no immediate parent and which can only get an immediate parent in the circumstances when the constraint is satisfied. An equation which states the value of the constrained feature must introduce a node which the former node can equate with, and which has an immediate parent. As long as the equation of these nodes can only occur in the right cases, the completeness requirement that all nonroots have immediate parents will only be satisfied if the constraint is satisfied.

Figure 21 gives an example of how a constraint equation can be simulated in SUG. The node on the right has the features *type: constraint* and *feature: case* to ensure that it will only equate with nodes which are also either enforcing or satisfying a constraint on the *case* feature. The feature *location* restricts the set of nodes which can be equated with to only those enforcing or satisfying a constraint on this instance of this *case* feature, as discussed further below. By also stating that the *case* feature has the value *nom*, any node satisfying this constraint must also specify the value to be *nom*.

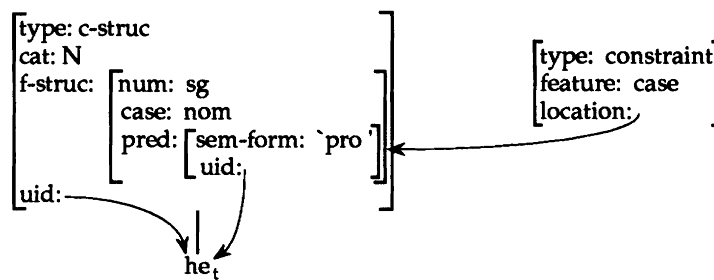


Figure 21: The simulation of (4), repeated below.

(4) he: N, $(\uparrow \text{NUM}) = \text{SG}$
$(\uparrow \text{CASE}) =_c \text{NOM}$
$(\uparrow \text{PRED}) = \text{'PRO'}$

Figure 22 gives an example of how an equation which might be needed to satisfy a constraint is expressed. This is the same as figure 18 except an additional node has been added. Like the above constraint node, it has the features *type: constraint, feature: case*, and *location* to restrict what constraints it can satisfy. Unlike the above constraint node, it has an immediate parent. Thus this node in no way interferes with any derivation which could occur without it, but if a constraint enforcing node exists for this feature in this location, then equating these nodes will allow the derivation to finish successfully. These constraint satisfying nodes must be introduced for any equation which gives a value for a feature which has a constraining equation for it somewhere in the grammar.

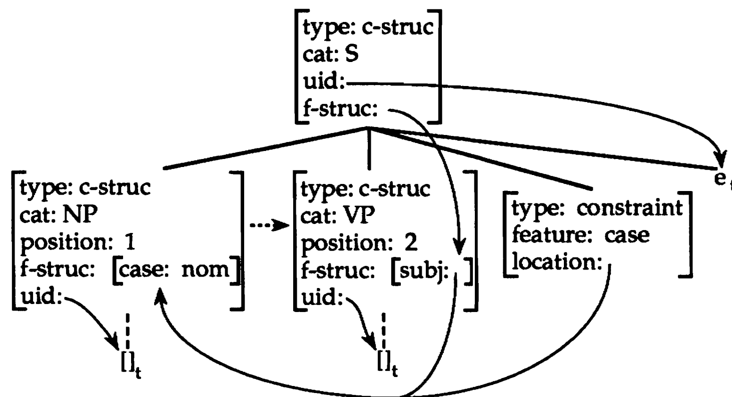


Figure 22: A partial simulation of (10) given that the feature *case* may be constrained. This is figure 18 with a constraint satisfying node added.

The use of the feature *location* in these constructions is not completely foolproof. It is possible that two constraint nodes which are not for the same f-structure to equate, thus forcing their f-structures to equate, without anything else ruling out the derivation. This is why every LFG′ grammar must exhibit unique association. The mapping which unique association guarantees to exist can be used to specify a *uid* feature with its value coreferenced with a terminal, for every f-structure. Thus these unwanted equations can not occur, since the *uid* features of the two f-structures could not unify. For our purposes the *uid* feature in each *pred* feature will suffice for this purpose, since all the f-structure in the examples in this paper always eventually get a *pred* feature, as discussed in section 3.2.1.3.

The construction above simulates constraint equations which specify a particular value which a feature must have. The other two kinds of constraining statements can also be simulated in this way. Existential constraints can be expressed simply by not specifying the

value of the feature when the constraining node is specified. The negation of an existential constraint can be expressed without a constraining node by giving the feature the value *none,* where *none* is not in the set of LFG′ atoms. In order to express the negation of a nonexistential constraining statement we must take advantage of the fact that there are a finite number of atoms other than semantic forms. Since LFG′ does not allow the *pred* feature to be constrained, the fact that there are an infinite number of semantic forms will never be a problem. Also, LFG′ only allows features which are negatively constrained to have atomic values, so no complex feature structures need to be considered as possible values for these features. Thus there are always only a finite number of values which a negatively constrained feature can have. Therefore a negative constraint can be expressed as the disjunction of the set of constraining equations specifying each nonexcluded atom as the features value, plus a negative existential constraint. This form of the constraint can be expressed using the mechanisms defined above.

This simulation of constraining equations is clearly ugly. However, this complexity in some sense reflects the complexity of enforcing constraining equations in general. Like with disjunction, this complexity can often be avoided by the clever use of ordinary feature value statements. For example, if we know that all NP nodes will somehow receive a value for the feature *case*, then simply stating the value of the *case* feature will have the same effect as using a constraining equation to restrict it to that value.

Simulating Nonlocal Feature Value Statements

Like the instantiation of immediate dominance metavariables, the instantiation of bounded dominance metavariables can be simulated by embedding the information predicated of their f-structures in the feature structure of a node. This requires the introduction of another type of node, since these metavariables must be distinguished from the immediate dominance metavariables. Figures 23 and 24 show how these nodes are used.

A node with *type: bounded* is introduced for every $\Downarrow$, as shown in figure 23. The subscript of the metavariable is specified as the *cat* of this node. The superscript of the metavariable determines what nodes are eligible to be the domain root. For every node in the rule which is a possible domain root, a structure is constructed with the bounded node dominated by it. This domination relation ensures that the bounded node will only equate with nodes in places dominated by the domain root. The other constraints on the control domain will be addressed below. The information predicated of this metavariable's f-structure is put in

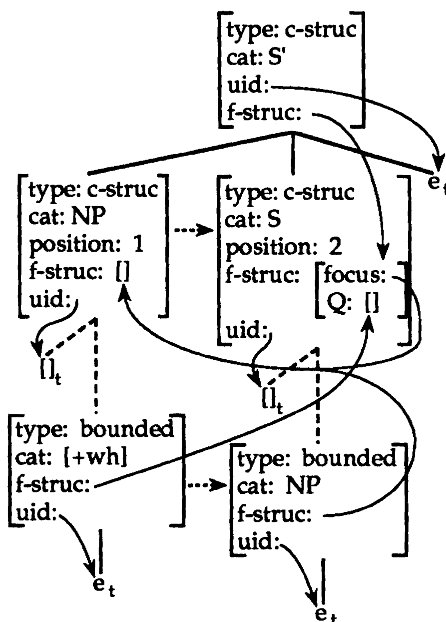


Figure 23: A partial simulation of (7), repeated below.

$$\text{(7) (K's 141) } S' \longrightarrow \begin{array}{c} NP \\ (\uparrow Q) = \Downarrow^{NP}_{[+wh]} \\ (\uparrow \text{FOCUS}) = \downarrow \\ \downarrow = \Downarrow^{S}_{NP} \end{array} \quad \begin{array}{c} S \\ \uparrow = \downarrow \\ \\ \\ \end{array}$$

the *f-struc* feature of the bounded node. Each bounded node also has an empty terminal as a child and a *uid* feature with this terminal as its value. This will cause this bounded node to satisfy the completeness constraints for a bounded node for a $\Uparrow$ in the same way the structure for a lexical entry satisfies the completeness constraints for an unexpanded preterminal node. The fact that these nodes do not have immediate parents ensures that they must find a bounded node for a $\Uparrow$ to equate with.

Figure 24 demonstrates how $\Uparrow$'s are represented. Like in the previous figure, the CAT feature ensure that the metavariable this metavariable is controlled by will have the proper subscript. The *f-struc* feature also works as above. This node is the complement of those just discussed in that it has an immediate parent, but its *uid* feature is an underspecified terminal. Thus if a $\Downarrow$ metavariable's node equates with this $\Uparrow$ metavariable node, the result will fulfill the completeness requirements for both nodes.

So far the only part of the bounding convention which has been enforced is that a node

(8) (K's 135) NP $\longrightarrow$ e
$\uparrow = \Uparrow_{NP}$

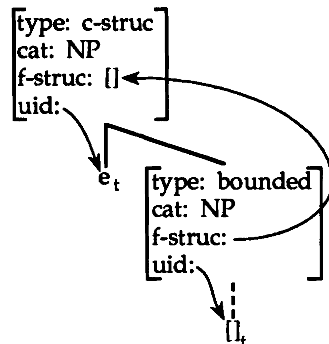


Figure 24: A partial simulation of (8), repeated nearby.

in a control domain must be dominated by the control domain's root. It still remains to incorporate the effects of bounding nodes. This can be done by adding another feature to bounded nodes, as shown in figures 25 and 26. The *domain* feature is set so that all the nodes which are not separated by a bounding node have the same value for it. Such a set can be defined simply by, in each rule, coreferencing the *domain* values of the parent and the child if the child is not a bounding node. The *domain* value of a bounding node is given an instance unique value, thus distinguishing the *domain* value of the set of nodes this node bounds from above from the other coreferenced *domain* values[17]. Now the bounded nodes can be given a *domain* feature with its value coreferenced with that of either its domain root or the node it annotates, depending on its metavariable. In this way two bounded nodes can equate only if the one for the $\Uparrow$ is not separated from the domain root of the one for the $\Downarrow$ by any bounding nodes.

The simulation of nonlocal feature value statements in SUG is interesting for two reasons. It demonstrates the importance of dominance relations in handling long distance dependencies, and it shows how more complicated restrictions on long distance dependencies can be encoded in SUG. The introduction of a special kind of node (i.e. bounded nodes) to handle these dependencies does not seem to be necessary for natural language. The simulation of (8) shown in figure 24 simply replaces a c-structure node's need for a *uid*

[17]In order for this method to be foolproof, we must also make provisions for the case when the complete sentence's root is not a bounding node. This can be done by arranging for a particular structure to always be the root of any complete structure, and providing a *domain* value from there, if necessary. I will not go into the details here.

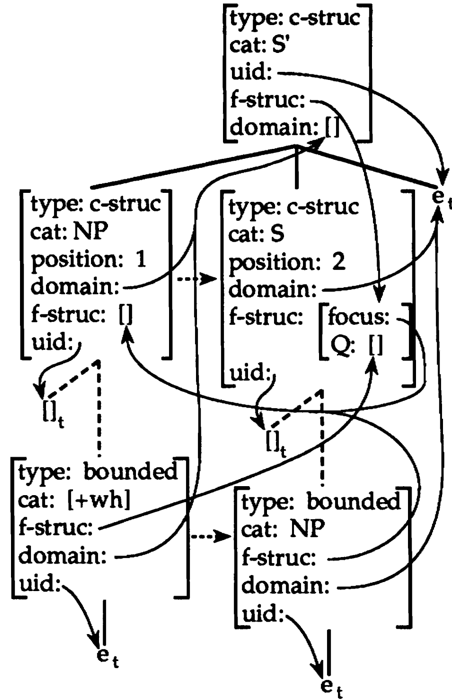


Figure 25: The complete simulation of (9), repeated below. This is figure 23 with the *domain* feature added.

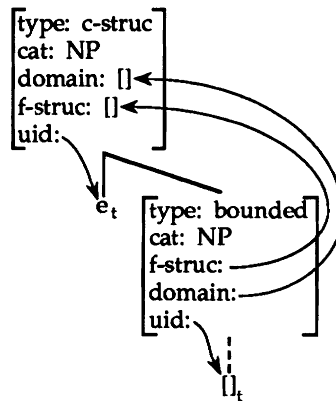


Figure 26: The complete simulation of (8). This is figure 24 with the *domain* feature added.

$$\text{(9) (K's 150) } S' \longrightarrow \begin{array}{c} NP \\ (\uparrow Q) = \Downarrow^{NP}_{[+wh]} \\ (\uparrow \text{FOCUS}) = \downarrow \\ \downarrow = \Downarrow^{S}_{NP} \end{array} \quad \begin{array}{c} \boxed{S} \\ \uparrow = \downarrow \end{array}$$

terminal with a bounded node's need for a *uid* terminal. If this type of rule is the only use of $\Uparrow$ then these rules can be eliminated and the bounded nodes for $\Downarrow$s can be made c-structure nodes. This would be the same as the treatment given above in section 3.1. With this change, the *domain* feature would work the same way. In some sense the use of the *domain* feature to constrain where $\Downarrow$ nodes can equate is simply a technique to allow the stipulation of constraints on long distance extraction. However, such a specification may simply be a declarative manifestation of a processing strategy for matching fillers with gaps. Under this interpretation there is no need to have the constraints on long distance dependencies follow from other constraints on the grammar, since they are rooted in a different component of the language system. More complicated constraints on long distance extraction can be encoded using other systems of features similar to the *domain* feature.

3.2.2.3 Enforcing Global Constraints

The four global constraints in LFG′ are functional uniqueness, completeness, coherence, and proper instantiation. Functional uniqueness is enforced by the fact that feature structures are being used to represent f-structure. Both completeness and the unextended version of coherence can be enforced as follows. Each time a *pred* feature value is specified, these constraints are encoded as constraint equations, and these equations are simulated as discussed above[18]. The extension of coherence which requires the features *topic* and *focus* to have their values identified with features which are subcategorized for, can be enforced using a constraint which requires each of these features to have its value set equal to that for a $\Downarrow$. This constraint can be represented in the same way as an existential constraint for some feature, say *bound,* in the value of *topic* or *focus.* The constraint can then be satisfied, when this value is set equal to that for a $\Downarrow$, in the same way as specifying a value for the feature *bound,* but without specifying this feature.

All of the clauses of proper instantiation either are already enforced, or can be enforced by restricting the grammar. Each controllee will have exactly one controller because of the

[18]The fact that this is possible is pointed out in [Kaplan and Bresnan, 1982], p 212.

uid feature in the controller nodes. Since only one controller node for each $\Downarrow$ will be placed under a given node, there will be only one controllee per control domain for each controller. The requirement that there must be at least one control domain for each controller can be enforced in the grammar by requiring all structures for the rule to have at least one controller node per $\Downarrow$. That domain roots must be distinct can similarly be enforced by not allowing more than one controller node to be placed under the same node.

3.2.3 Discussion

The above constructions can be used to translate any LFG′ grammar to an equivalent SUG grammar. This demonstrates that LFG′ is at most as powerful as SUG. This translation is also interesting for other reasons. It demonstrates that many of the linguistic generalizations captured well in LFG can also be captured perspicuously in SUG. The components of LFG which are of particular interest in this regard are its explicit representation of predicate–argument structure and its treatment of long distance dependencies.

In addition to the representation of phrase structure in its c-structures, LFG has an explicit representation of predicate–argument structure in its f-structures. This permits phrase structure and predicate–argument structure to be expressed independently, thus freeing the phrase structure from the need to exactly mimic predicate–argument structure. LFG's f-structures can be expressed in SUG within the feature structures which label phrase structure nodes, with the f-structure of the root being the f-structure of the sentence. By specifying predicate–argument structure in this way, the correspondence between semantic constituents and syntactic constituents is maintained throughout a derivation, but these structures need not be identical. This permits the perspicuous representation of semantic relationships such as those in raising verbs like 'seems'. This method of expressing predicate–argument structure in SUG works independently of the methods LFG uses for specifying c-structure, and thus can be used in any SUG grammar. The analyses proposed above in section 3.1 are especially suited to these specifications because the domain of locality of the grammar entries is large enough to state predicate–argument relationships directly, without the feature passing techniques needed in LFG.

LFG's treatment of long distance dependencies provides fairly adequate mechanisms for specifying what dependencies can and can't exist. SUG can simulate these mechanisms using dominance relations and a simple system of feature constraints. The use of dominance relations to handle long distance dependencies was demonstrated in section 3.1, but those

analyses allowed long distance dependencies which are not found in English. The system of feature constraints used to simulate the bounding nodes of LFG can be applied to the SUG analysis in section 3.1 to help rule out these unwanted long distance dependencies. Other similar systems of feature constraints could be developed if this simple system proved inadequate to capture the desired constraints. One such system is discussed at the end of section 3.6 on Lexicalized Tree Adjoining Grammar.

3.3 Description Theory

Description Theory (D-Theory, [Marcus *et al.*, 1983]) solves several problems in the deterministic parsing of natural language by having the syntactic processor build a partial description of a sentence's phrase structure, rather than a complete specification. The development of SUG was heavily influenced by this work, as is evident from SUG's extensive use of partial descriptions in the specification of both nodes in the phrase structure and the structural relations themselves. After describing crucial aspects of D-Theory, this section will discuss how SUG's use of partial information allows it to adopt many of the parsing strategies advocated in D-Theory.

D-theory uses its partial specification of phrase structure to avoid specifying things which can only be determined later in the parse or with the use of semantic information. For example, the output of a D-Theory parser can leave unresolved ambiguities between possible prepositional phrase attachments and ambiguities arising from coordination. This is possible because D-Theory uses dominance rather than immediate dominance to specify trees, and because two nodes which are not equal may be equated at a later time. A prepositional phrase, for example, can be attached as high as possible, then lowered to the appropriate phrase when the disambiguating information is brought to bear.

The parsing framework which D-Theory uses is based on the Marcus Parser ([Marcus, 1980]). The basic data structures are a buffer for unattached constituents and a stack for incomplete constituents. There are a small set of operations which can be performed on these data structures, including attaching an item in the buffer to an item in the stack and dropping an item from the stack into the buffer. The grammar specifies when to perform each operation. The parser always proceeds deterministically, in the sense discussed in [Marcus, 1980]. This requirement dictates that once information is added to the state of the parse, it can not be removed. It is this indelibility of information that makes parsing

in the presence of ambiguity difficult.

A D-Theory grammar is specified using two mechanisms. One is a form of context free grammar and the other is a set of templates which trigger certain actions. Given an incomplete constituent in the stack, the context free rules say how that constituent can be completed. The templates have a pattern which includes information about the state of the parser. If this pattern matches the current state of the parser then the actions of the template are performed. For example, if the first buffer cell contains the word 'the' then the construction of an NP is triggered by pushing an NP on the stack and attaching 'the' below it. This mechanism alleviate the need for the context free grammar to have categories such as determiner, which do not fit well into many characterizations of phrasal categories. The idea behind using these two separate mechanisms is that the leading edges of phrases are relatively easily recognized and thus can be handled with simple patterns.

SUG does not assume any particular parsing framework, but the fact that it could be used as the declarative portion of an investigation into deterministic parsing is evident from SUG's relation to D-Theory's declarative portion. SUG is powerful enough to express anything expressible in the portion of D-Theory which is not procedural. Perhaps more importantly, SUG has the properties which are important in D-Theory for parsing deterministically.

D-Theory's ability to partially specify information about phrase structure is shared by SUG. In particular, SUG has the ability to express dominance relations and the ability to postpone the equations of nodes, thereby allowing SUG to express the ambiguities discussed above. The difference between SUG's approach to this underspecification is that of D-Theory is that D-Theory considers a parse complete even without resolving the ambiguities. D-Theory assumes that later processes, like the semantic component, take the result of the syntactic processor and further disambiguate it. SUG, on the other hand, requires that the structure be completely disambiguated when the parse is done. However, SUG is not intended to be a model of the syntactic component of a parser alone. Disambiguations due to semantic influences would be included in the SUG parsing process. If there are situations in which people never disambiguate between some possibilities, then a notion of partial SUG parse could be defined which would allow for this underspecification, but the exact nature of these partial parses would have to be constrained to maintain the basic semantics of SUG grammars.

The mechanisms D-Theory uses to specify grammatical information are easily translated

into SUG. Context free grammar rules can be specified as outlined in section 3.2.2.1. The use of templates to trigger constituents can be expressed by adding the structure which the action would create to the grammar. The presence of the template contents in this structure limits its use to the appropriate contexts. The structure given to 'the' in figure 6 illustrates this basic idea. The use of such nonhead projection of nodes will be discussed further in the next section.

3.4 Abney's Licensing Parser

In [Abney, 1986], Abney presents a parser which is designed to be a model of linguistic performance while still reflecting some concepts from Government-Binding Theory (GB). It is interesting to compare this parser to Structure Unification Grammar because it is a procedurally defined investigation into language and because the concerns driving its design were more computational than the other systems discussed in this paper. The central concept in Abney's parser, that of licensing, is important because it not only manifests important linguistic generalizations, but it is also easily parsable. Licensing relations are easily expressed in SUG in ways which preserve their usefulness in parsing. This section will first describe Abney's parsing system, then discuss how the insights from this investigation can be manifested in SUG.

3.4.1 The Parser

The central concept in Abney's parser is that of licensing. Licensing is a generalization of θ-assignment in GB. Essentially, a phrase is licensed if it has some function in the structure of the sentence. Thus, not only do NP's have to find a θ-grid position to fill, all other phrases have to find an analogous role. Abney chooses licensing relations as the central concept of his parser because they are both easily parsed and very general across languages. With the exception of their directionality, many licensing relations seem to be language universal[19].

[19] In the sense that licensing relations are expressions of language universal thematic structure, they have a lot in common with Lexical Functional Grammar's f-structure. However they differ from f-structure in that they are direction specific and are more tightly constrained to conform the phrase structure relationships.

3.4.1.1 The Parser's Representations

The parser does not actually build phrase structure; it builds licensing structure. However, because of the restrictions placed on licensing relations, there is always a simple mapping from licensing structure to phrase structure. A licensing relation is a ternary relation between the licensing node, the licensed node, and the role associated with the relation. Like θ-roles, licensing relations are unique, in the sense that each node is only licensed by one relation. Also like θ-roles, they are determined by information associated with lexical entries (i.e. the heads of phrases). Unlike θ-roles but like Case assignment, licensing relations are directional, in that they can only hold when the licensor and licensee are in a specified order. This direction is specific to each licensing relation. Licensing is also restricted to hold between sisters in the phrase structure tree[20]. The only additional restrictions needed to ensure that any licensing structure has an associated phrase structure are that licensing relations be nonreflexive and acyclic, which are independently desirable constraints. Given this relationship between licensing structure and phrase structure, I will talk of licensing structures in their more familiar phrase structure form.

Information about licensing relations are specified as triples associated with lexical entries. A triple specifies the direction in which it must be assigned, the type of node to which it must be assigned, and the role of this licensing relation. A set of such triples will be called a licensing frame. These frames are carried along with a word in the parse and determine what licensing relations can be assigned.

There are a couple of problems with parsing using licensing relations which have prompted Abney to add another mechanism for specifying them. Many of the things licensed by a given head are adjuncts. If we want the parser to be efficient, it should not have to be looking for every adjunct which might modify a phrase. Thus it is desirable to specify the licensing relations for adjuncts on the adjuncts themselves. The other reason for doing this is to facilitate the detection of failed parses early in the parse. By specifying what will license a prehead adjunct on the adjunct, it is possible to tell if it can be incorporated into the parse by seeing if the expected licensor can be licensed by something in the current

[20]Since the phrase structure tree is defined in terms of the licensing structure, this is actually not a restriction but simply a definition of how the phrase structure tree relates to the licensing structure. It is significant, however, in that Abney wants his analyses to parallel those in GB, and in this way it restricts the licensing relations he can propose. As a specific example, he can not say that 'wants' licenses 'Mary' in the sentence "John wants Mary to leave" without violating the GB analysis of 'Mary' as being a constituent of "Mary to leave".

structure. A similar technique can be used for prehead subcategorized arguments, such as the subjects of subordinate clauses. In this way a failed parse can be detected as soon as it can not be seen how something will be incorporated into the parse. These additional licensing specifications are called anti-relations. They have the same structure as licensing relations.

3.4.1.2 The Parsing Process

Since Abney views the parser as a processing model of language, it proceeds incrementally from the beginning of the sentence to the end and only recovers one parse at a time. If it can't be seen how the next word will be incorporated into the current parse structure, the parser will stop and fail. If the sentence is ambiguous it disambiguates the sentences in a way which reflects people's preferences.

The state of a parse is represented as a list of partial subtrees, one of which is distinguished as the current subtree. When a word is read, it is added to the end of the list, after being projected to its maximal projection. A list of pattern-action rules is then consulted, and the first pattern which matches has its action done. These patterns are restricted to only refer to the root of the current subtree and an unspecified part of the near edge of the preceding subtree. The actions can combine and add information to the subtrees, and can modify a small amount of the information already specified in them.

Abney uses several actions in his parser. The most important one is attachment. If the root of a subtree matches the restrictions for a licensing relation on an adjacent subtree, then it can be attached to that subtree, thus filling that licensing relation. This operation can also be done when the root of a subtree has an anti-relation which matches a node on the near frontier of an adjacent subtree. Since he is doing these attachments whenever he can, sometimes a choice is made which must later be undone. As mentioned above, these changes are limited. One such action, called REANALYZE, is used when the wrong lexical entry for a word has been chosen. This action detaches a previously attached projection of a word and replaces it with a homonym. This can only be done if nothing has been attached under the replaced projection. Another mutating action is called STEAL. It detaches an argument from one subtree so as to attach it to some other subtree. A third mutating action, called REPLACE, detaches one node so as to attach another in its place. The final such action which he discusses, called frame switching, replaces one licensing frame with another for that word, as long as the arguments which have already been attached have

analogous positions in the new frame. These mutating actions are what prevents this parser from being deterministic, in the sense of [Marcus, 1980].

During parsing it is often the case that there is more than one way to attach a constituent. Since Abney only wants to get one parse at a time, he has to choose one of these attachments. He does this by ordering the possibilities as follows ([Abney, 1986], pp12).

1. θ-licensers preferred over non-θ-licensers
2. Verbs preferred over other categories
3. Low attachment preferred

These attachment preferences reflect the disambiguation choices people make.

The final component of Abney's parser is a mechanism for placing empty categories, but this component is not central to his investigation. Abney considers his parser to be one component of a complete parsing model, namely the one which recovers licensing relations. Thus he is not concerned with long distance dependencies. However, he implements a mechanism for placing traces in order to detect where empty categories fill licensing relations. This mechanism is analogous to slash passing in GPSG, which is not surprising since both GPSG and Abney's parser can only specify immediate dominance relations between nodes. When a wh-element is encountered a trace is created and appointed a "host". As the parse proceeds the trace is passed from parents to children until a node is found which can license it. This way of finding gaps enforces that a moved element must c-command its trace, but no other constraints on movement are embodied.

3.4.2 Comparison with SUG

The most important insight Abney makes in this investigation is the concept of licensing. Recovering the licensing relations in a sentence is a large part of parsing, yet when these relations are represented well they are easily recoverable. One important aspect of Abney's representation of licensing relations is its partiality. For example, the specification of licensing relations Abney uses could be translated into a context free grammar, but such a representation would not be adequate for incremental parsing. Because of its partiality, SUG's representation also allows for the flexible specification and manipulation of licensing information. Both types of licensing relations can be represented in SUG using the phrase structure relations they imply. For example in figure 6, 'rolls' expresses its licensing of a

subject by specifying an *idom* link from its maximal projection to the subject $\overline{\overline{\text{N}}}$, and 'quick' expresses its anti–relation by specifying an *idom* link from its licensing $\overline{\text{N}}$ to its maximal projection. In both cases underspecified terminals are used to express the obligatoriness of these licensing relations, and words are used to specify the uniqueness of these licensing relations.

The need for representing licensing relations in both regular relations and anti–relations is another important insight of Abney's licensing parser. There are several phenomena, such as movement restrictions, which point to a distinction between subcategorized arguments and adjuncts. The usefulness of anti–relations in Abney's parser is further evidence for treating these two types of licensing with distinct mechanisms. As shown above, SUG has the ability to express both these syntactic relationships. In SUG as in Abney's parser, the difference between expressing regular licensing relations and anti–relations is that a regular licensing relation is specified with the licensor, while an anti–relation is specified with the licensee. This means a regular licensing relation is specified in SUG as an *idom* relation from a headed parent to a headless child, while an anti–relation is specified in SUG as an *idom* relation from a headless parent to a headed child. The distinction between these two types of structures was mentioned previously in section 3.1.1.

The differences between specifying a constituent relationship with the head of the parent and specifying it with the head of the child may explain many of the differences between languages with more fixed word order and little case marking, such as English, and languages with freer word order and rich case marking, such as Warlpiri. In the former case most constituent relationships are determined by the head of the parent, and thus are specified as an *idom* relation between a headed parent and a headless child. This necessitates some information such as word order constraints in order to determine which nodes correspond to which argument positions. However, the arguments do not need to be explicitly marked, since the portion of the structure which they contribute is the same regardless of what position they fill. In the case of languages like Warlpiri, constituent relationships are determined by the head of the child. This means that word order constraints are unimportant, since each argument carries in its structure information which specifies what argument slot it fills. However, because the portion of the structure which a word contributes is dependent on what argument slot it fills, this information must be explicitly marked on the word itself[21].

[21] For more discussion of this point, see [Brunson, 1988].

3.5 Tree Adjoining Grammar

The extensive amount of work which has been done on the formal characteristics and linguistic applications of Tree Adjoining Grammar (TAG, [Joshi, 1987a], [Vijay-Shanker, 1987], [Joshi *et al.*, forthcoming, 1990], [Kroch and Joshi, 1985]) make it well worth discussing here. This is especially true given the similarity between SUG and TAG. The basic objects of both TAG and SUG are phrase structure trees, thus permitting a distinction between phrase structure and derivation structure. Also, the size of SUG's domain of locality for expressing grammatical information is very similar to TAG's.

This section will be primarily concerned with showing that any TAG grammar can be translated into an equivalent SUG grammar. The translation which will actually be given is between Feature Structure Based Tree Adjoining Grammar (FTAG, [Vijay-Shanker, 1987]) and SUG, because it is more straight forward. An independently desirable restriction on FTAG makes it equivalent to TAG. After these two versions of TAG are defined and the translation to SUG is given, the implications of this translation will then be discussed, with particular attention given to the work which has been done on the formal power of TAG. Most of the discussion about the linguistic work which has been done in TAG will be postponed until the next section, which will discuss Lexicalized Tree Adjoining Grammar.

3.5.1 The Definition of TAG and FTAG

As mentioned above, the objects used in a Tree Adjoining Grammar derivation are trees. The trees in a TAG grammar are called elementary trees and are of two kinds, initial trees and auxiliary trees. The initial trees represent simple structures. The auxiliary trees represent the recursive components which can be inserted into the simple structures to produce arbitrarily large structures. The insertion is done using a process called adjunction. Feature Structure Based Tree Adjoining Grammar works the same way as TAG, but the adjunctions in FTAG are constrained with a slightly different mechanism than in TAG.

3.5.1.1 TAG's Definition

Formally, a TAG grammar is a tuple $G=\langle V_N, V_T, S, I, A\rangle$ where V_N is a finite set of nonterminals, V_T is a finite set of terminals, S is a distinguished nonterminal, I is a finite set of initial trees, and A is a finite set of auxiliary trees. An initial tree has S labeling its root, elements of V_T labeling its leaves, and elements of V_N labeling its internal nodes.

An auxiliary tree also has elements of V_N labeling its internal nodes, but its frontier also contains one element of V_N. All the other leaves are labeled with elements of V_T. If this one nonterminal leaf, called the foot node, is labeled with $A \in V_N$, then the root is also labeled with A.

In a TAG derivation, trees are combined using tree adjunction, as depicted in figure 27. To adjoin a tree T_2 at a node x in a tree T_1, T_1 is split at x and T_2 is inserted between the two pieces. More precisely, the subtree of T_1 below x is excised, T_2 is substituted in its place, and the excised subtree is substituted for the foot node of T_2. The nonterminal labeling x must be the same as the one labeling the root and foot nodes of T_2.

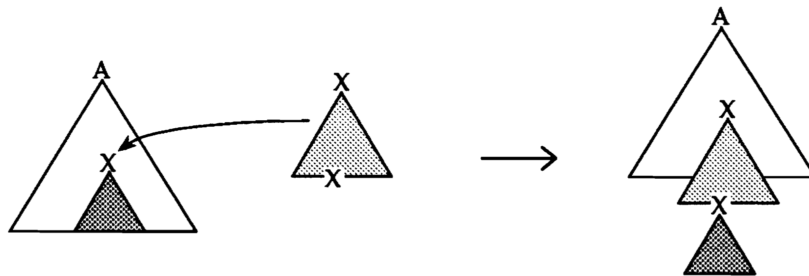


Figure 27: The adjunction operation in TAG.

In addition to the limits nonterminal labels place on possible adjunctions, each of these possible adjunction sights have adjunction constraints. A node's constraints specify what auxiliary trees can be adjoined at that node. If no trees can adjoin then that node has a null adjoining (NA) constraint. Adjunction constraints can also specify that a node has an obligatory adjunction (OA) constraint. An OA constraint requires that the node have some tree adjoined at it before a derivation using the tree is finished.

A TAG derivation uses one initial tree, on which a finite number of adjunctions are performed. The resulting tree must have no remaining OA constraints. The tree set $T(G)$ generated by a TAG grammar G is the set of all trees which are the results of derivations using trees from G. The string language $L(G)$ generated by G is the set of strings which are yields of trees in $T(G)$.

3.5.1.2 FTAG's Definition

A FTAG grammar is the same as a TAG grammar, except that nodes which are labeled with nonterminals also have two feature structures associated with them, a top feature structure

and a bottom feature structure. These feature structures take the place of adjunction constraints, which do not exist in FTAG. When a tree T_2 is adjoined at a node x in a tree T_1, the trees are combined as in TAG, except the top feature structure of x must unify with the top feature structure of the root of T_2 and the bottom feature structure of x must unify with the bottom feature structure of the foot of T_2. After being so unified, the root and foot keep their feature structures in the resulting tree. This operation is illustrated at the top of figure 30. When the derivation is complete the top and bottom feature structures of each node must unify. Obligatory adjoining constraints can be simulated in this system by giving a node top and bottom feature structures which can not unify with each other, thus forcing something to be adjoined at that node to separate the inconsistent feature structures.

3.5.2 Expressing FTAG in SUG

To translate an FTAG grammar into an SUG grammar, the SUG grammar must allow sets of equations which simulate all the possible adjunctions within the FTAG grammar, and the SUG grammar must be constrained so that these are the only possible sets of equations. This section will proceed by first explaining what the translation is and how the resulting SUG grammars can simulate FTAG derivations, then it will be shown that these simulations are the only possible derivations in the SUG grammars.

3.5.2.1 The Translation

To simulate adjunction in an SUG grammar, it must be possible to insert an arbitrary amount of structure at each adjunction sight. To allow this, each FTAG node which is a possible adjunction sight is mapped to a pair of nodes in the SUG grammar called twins. As shown in figure 28, one of these nodes includes the top feature structure of the FTAG node, the other includes the bottom feature structure of the FTAG node, and the former dominates the later. Since there is only a dominance relation between twins, they can be pulled an arbitrary distance apart to allow another structure to be inserted between them, thus simulating an adjunction. If no adjunction is simulated at these nodes, then the dominance relations allows the twins to be equated, thus unifying the top and bottom feature structures as required for an FTAG derivation to finish.

The *uid* and *twin* features restrict the possible sets node equations to those which simulate FTAG derivations. The need for either an adjunction or the unification of the top

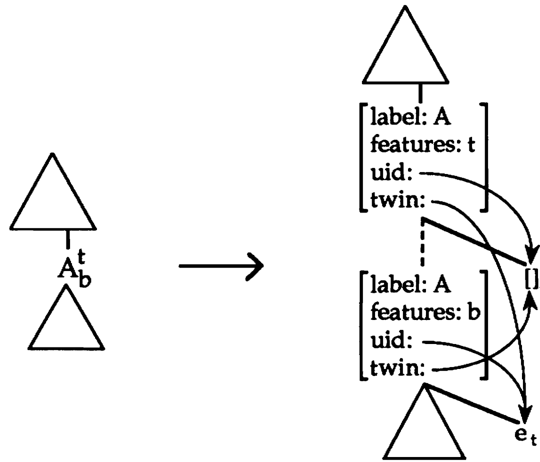


Figure 28: The translation of a nonterminal in a TAG tree into two nonterminals in a SUG structure. The dominance link between the SUG nodes allows the top and bottom halves to be separated by a simulated adjunction or to be equated if there are no adjunctions at this node.

and bottom feature structures at this node is reflected in the top twin's underspecified *uid* terminal and the bottom twin not having an immediate parent. Because all nodes have the *twin* feature, the equation of two nodes will always force the equation of their respective twins' *uid* terminal, and thus their respective twins. This can only happen if a node equates either with its own twin, thus simulating the feature unification, or with the top and bottom nodes shown in figure 29, thus simulating an adjunction, as discussed below.

The roots and feet of auxiliary trees require additional nodes to be added to the corresponding SUG trees. Each root and foot node gets mapped to a pair of twin nodes to permit adjunction there, but there is an additional node above the twins for the root and an additional node below the twins for the foot, as shown in figure 29. These additional nodes are what equate with the twins of the adjunction sight when this auxiliary tree is adjoined, as demonstrated in figure 30. The extra root node equates with the top of the two twins and the extra foot node equates with the bottom twin. The *features* features of these two extra nodes are coreferenced with those of their nearest twin node to ensure that the top and bottom feature structures are unified as required for FTAG adjunctions.

The only remaining aspects of FTAG grammars which need to be mapped to SUG grammars are terminals and tree structure. FTAG terminals are simply mapped to SUG

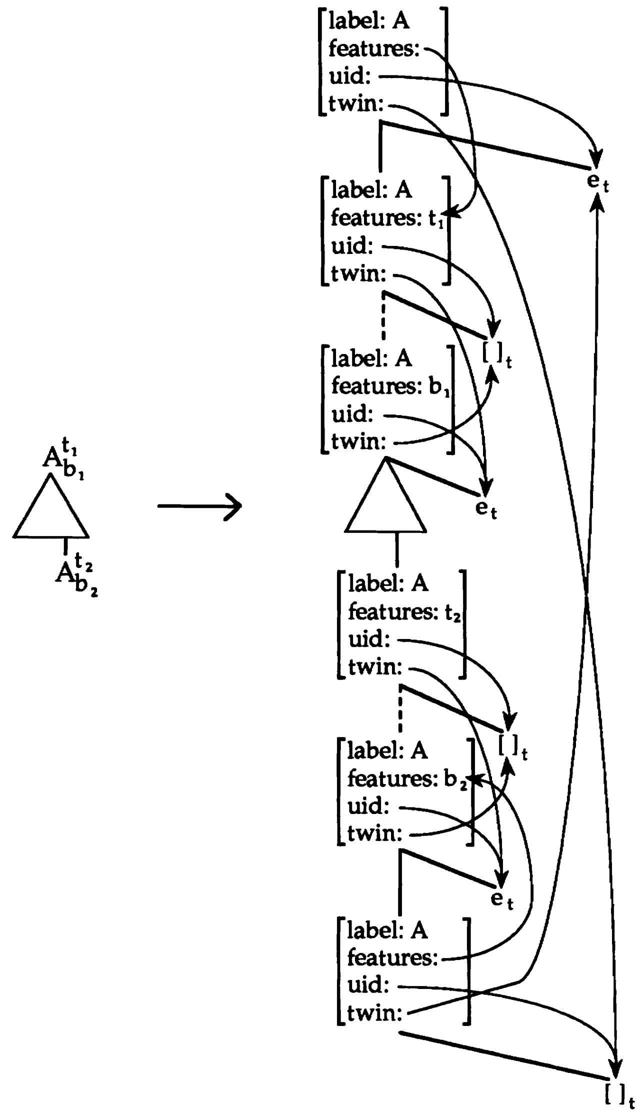


Figure 29: The translation of root and foot nodes in a TAG tree into two nodes each for simulated adjunctions at these nodes, as given in figure 28, and one pair of nodes for simulated adjoining of this tree. These later nodes are the top and bottom nonterminals in this figure.

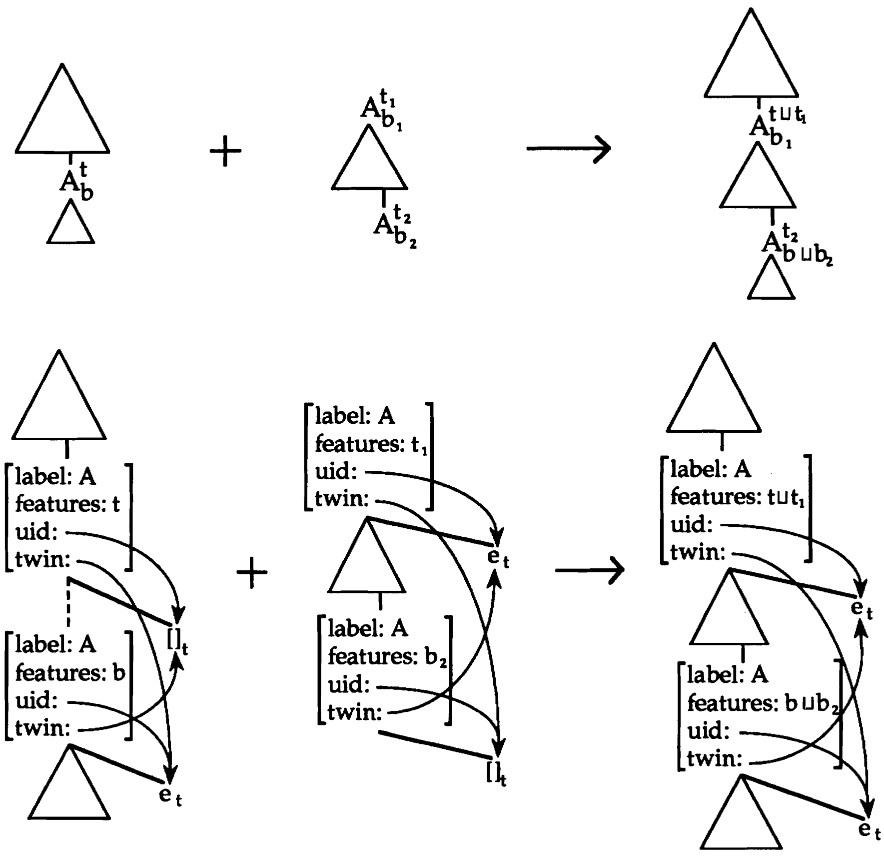


Figure 30: The simulation of a TAG adjunction in the equivalent SUG grammar. After the simulated adjunction the nodes equated can not be involved in any more equations, but further adjunctions at the TAG nodes can be simulated with the other four nodes shown in figure 29, which are not shown here.

terminals. Each immediate dominance link in a FTAG tree is mapped to one in the associated SUG tree as was shown in figure 28. Links from nonterminals to terminals are translated into links from the bottom twin of the nonterminal to the terminal. Links between nonterminals are translated into links from the bottom twin for the upper node to the top twin of the lower node. Ordering relationships are translated equally transparently; if one node in an FTAG tree precedes another, then all the nodes in the SUG structure associated with the former node linearly precede all the SUG nodes associated with the later node.

3.5.2.2 The Proof of Equivalence

To demonstrate that the SUG grammars which result from the above translation are weakly equivalent to their associated FTAG grammars, it will be shown that the only sets of equations which can occur in these SUG grammars are those which simulate derivations in the FTAG grammars. These constraints in the SUG grammars are primarily accomplished with the *uid* (for Unique IDentification) and *twin* features. All nonterminals have a *uid* feature whose value is a terminal which the nonterminal immediately dominates. If this terminal has its word specified, then the nonterminal can not equate with any other node with its *uid* terminal's word specified, because words are instance unique. If a node's *uid* terminal does not have its word specified, then before the derivation is over this terminal must equate with a terminal with its word specified. Since terminals can only equate as a side effect of nonterminal equations, in order for the derivation to finish the nodes with underspecified *uid* terminals must equate with nodes whose *uid* terminals have their words specified.

The *twin* feature coordinates the two equations needed to simulate an adjunction. In twin nodes the value of the *twin* feature is the *uid* terminal of its other twin[22]. In the additional nodes for the root and foot nodes of auxiliary trees, the value of the *twin* features are the *uid* terminals of the other additional root or foot node in the tree. Thus when the top node of a pair of twin nodes equates with an additional root node, the bottom node of these twins must equate with the additional foot node, and vice versa. After this pair of equations both the nodes have their *uid* terminal filled. Through the *features* features, these equations also unify what was the top feature structure of the adjunction sight with

[22]The value of the *twin* feature could be the other twin itself, but this would introduce unnecessary cycles in the feature structures.

what was the top feature structure of the root, and what was the bottom feature structure of the adjunction sight with what was the bottom feature structure of the foot. If these unifications fail then the equations are not possible, as desired. If a pair of twins are not used to simulate an adjunction, then they can be equated to each other, thus simulating the unification of the top and bottom feature structures as necessary for an FTAG derivation to finish, and filling the upper twin's *uid* terminal.

Now that I have described how the SUG grammars can simulate the operations in their associated FTAG grammars, let me convince you that no other sets of equations are possible in these SUG grammars. There are five kinds of nodes, top twins, bottom twins, additional roots, additional feet, and terminals. Let us consider all the possible equations which could occur between these kinds of nodes. First, terminals can only equate with other terminals and only as a side effect of the equation of nonterminals, so this case is subsumed under the other cases. Bottom twin nodes and additional root nodes can never equate with each other because their *uid* features conflict. Top twin nodes and additional foot nodes can never equate to each other because their *twin* features conflict. Additional root nodes can never equate to additional foot nodes because their *twin* features would cause the *uid* terminals of their associated additional foot and root nodes to equate, thus forcing these associated nodes to equate, thereby forming an unallowable cycle in the tree structure. All the remaining types of equations, namely between top and bottom twins, top twins and additional roots, or bottom twins and additional feet, are possible and correspond to combinations described in the previous paragraph.

To show that the derivations in the SUG grammars will all correspond to valid derivations in their FTAG grammars, I still need to show that the SUG derivations can only finish if they correspond to finished FTAG derivations. Since an SUG derivation can not stop unless all the terminals have their words specified, the *uid* features ensure that the SUG derivations here can only finish when all twin nodes have either been involved in an adjunction or have been equated to each other. Also, the structures for auxiliary trees can only be used in a derivation if they are used in an adjunction, since the word of the additional foot node's *uid* terminal is not specified. Thus these grammars will never simulate an incomplete FTAG derivation.

Given that the derivations of the SUG grammars all correspond to FTAG derivations in their associated FTAG grammars, and vice versa, all that is needed to show that these associated grammars are weakly equivalent is that the sentences resulting from associated

derivations are the same. Since the two derivations have exactly the same derivation structure, this can be proved by induction on the steps of the derivations. The important point here is that the mapping from FTAG trees to SUG structures preserves all the tree and ordering relations in the FTAG trees. Each FTAG node maps to a set of SUG nodes which all have the same ordering constraints as the FTAG node and which together participate in the same tree relations with other such sets of SUG nodes as the FTAG node. This fact makes the base case of the induction easy, since the terminals in an FTAG elementary tree map directly to the terminals in the associated SUG grammar entry[23]. From figure 30 it should be clear that the tree and ordering relations resulting from an FTAG adjunction map in the same way just described to the SUG structure resulting from the simulation of this adjunction. Again, since terminals map to terminals and ordering constraints are preserved across this mapping, the yield of the FTAG tree which results from an adjunction is the same as the yield of the SUG structure which results from the simulation of the adjunction. The only other operation, that of equating top to bottom feature structures, does not change the yields of the structures in either case. Thus by induction on the steps of a derivation, the sentence generated by an FTAG derivation is the same as the sentence generated by the associated SUG derivation.

3.5.3 Discussion

The transformation given above demonstrates that Structure Unification Grammar is at least as powerful as Feature Structure Based Tree Adjoining Grammar. Since FTAG is known to be undecidable, this implies that SUG is undecidable, as is any formalism which combines the unrestricted use of feature structures with the ability to generate arbitrarily large structures ([Vijay-Shanker, 1987]). However, if we restrict the feature structures in grammars so they can not grow arbitrarily large in a derivation, then both FTAG and SUG become decidable[24] ([Vijay-Shanker, 1987] for FTAG). All the SUG grammar entries mentioned in this thesis have this property. This restriction makes FTAG equivalent to TAG, but SUG under this restriction is strictly more powerful than TAG, since SUG can recognized the language $a_1^n a_2^n \ldots a_m^n$ for any fixed m and TAG can only do this for m less than five ([Vijay-Shanker, 1987]).

[23]When comparing the yields of FTAG trees with SUG structures I will not include the empty terminals which the SUG structures use for their *uid* feature values.

[24]There are other ways to make SUG decidable. In particular, if we require that the grammar be lexicalizable, then it will be decidable (same argument as [Schabes *et al.*, 1988]).

Several other formalisms have been proven weakly equivalent to TAG, and thus are strictly less powerful than SUG. These formalisms include Combinatory Categorial Grammars, Head Grammars, and Linear Indexed Grammars. The fact that SUG can express all the languages expressible in Combinatory Categorial Grammar is of particular interest here, as this formalism will be discussed later in this chapter. This level of expressive power demonstrates that SUG is capable of expressing a very broad class of grammatical constraints.

The most important characteristic which makes TAG linguistically interesting is its large domain of locality. This domain determines what constraints can be expressed locally within a single grammar entry. TAG has the ability to express long distance dependencies within its domain of locality; both a gap and its filler can be specified in a single grammar entry. The dependencies can stretch over an unbounded distance through adjunctions. Examples of such trees are given for Lexicalized Tree Adjoining Grammar in figure 32. Since the above transformation maps each FTAG elementary tree to an equivalent SUG grammar entry, SUG has at least as large a domain of locality as TAG. An example of expressing long distance dependencies was already given in figure 7, and more will be given in the comparison of SUG with Lexicalized Tree Adjoining Grammar.

Another important characteristic of TAG is the fact that it represents phrase structure explicitly, rather than, for example, using the derivation structure of a CFG as does LFG. This both facilitates the expression of constraints in the grammar and allows a distinction to be made between phrase structure and derivation structure. SUG also represents phrase structure explicitly. The distinction between phrase structure and derivation structure will be important in the discussion of Combinatory Categorial Grammar (CCG). CCG espouses a type of phrase structure which is quite different from traditional views of phrase structure, but which does a good job of capturing regularities in coordination. I will argue that the structures espoused by CCG are best thought of as derivation structures in SUG, thus allowing the advantages of traditional phrase structure to be kept while still capturing the conjunction generalities in the derivation structure.

3.6 Lexicalized Tree Adjoining Grammar

Lexicalized Tree Adjoining Grammar (LTAG, [Schabes, 1990]) adds to TAG a substitution operation. This does not increase the power of the formalism, but it allows more flexibility in

the specification of grammar entries. As a result of this increased flexibility, TAG grammars can be translated into lexicalized grammars in LTAG. The resulting grammars are very similar to those which have been presented in this paper for SUG. This section will show that grammar specification in SUG is flexible in the ways it is in LTAG, and show that where they differ SUG is actually more flexible. I will also briefly discuss the linguistic reasons why this extra flexibility may not be desirable and how this constraint can be expressed in SUG.

The major motivation for adding more flexibility in the specification of TAG grammar entries is to allow them to be the minimal structures which localize semantic and syntactic dependencies. Like TAG elementary trees, LTAG elementary trees have a large enough domain to locally express syntactic and semantic relationships, such as long distance dependencies and predicate-argument structure. In addition, the substitution operation in LTAG makes it possible for each elementary tree to contain only one predicate-argument structure, since subconstituents can be substituted in. These properties allow LTAG elementary trees to each be associated with a particular lexical item, called the anchor, which is the source of the syntactic and semantic information in the tree[25]. These structures are semantically minimal in the sense that their meaning is not best thought of as the composition of smaller meanings. The lexicalization of a grammar facilitates parsing because only the portion of the grammar which is pertinent to the words in a sentence need be considered in parsing the sentence. It also results in a more modular representation of the grammar.

SUG also allows grammar entries to be minimal in the sense just discussed. As was discussed in the previous section, SUG's domain of locality is sufficient to locally express long distance dependencies and predicate-argument relations. LTAG's division of information among grammar entries is also possible in SUG, since the substitution operation of LTAG is just another example of node equation in SUG. This ability to divide information is demonstrated in the lexicalized grammar entries given in section 3.1.

Figure 31 gives several simple LTAG elementary trees and SUG structures which could be used to express the same grammatical information[26]. As explained in section 3.1, the fact that the substitutions are mandatory is expressed in SUG using underspecified terminals and the *head* feature. Note that the SUG versions of the adjuncts do not produce a Chomsky

[25]There are cases, such as idioms, where the anchor is actually more than one word. In the comparisons between LTAG and SUG given below I will assume the one word case, but the number of terminals in the structure does not effect any of the points made.

[26]Many of the LTAG examples used here are taken from [Schabes, 1990].

adjunction structure, as do the LTAG versions. Also note that the SUG version of the structure for 'thinks' treats the S object like any other subcategorized argument, unlike the LTAG version, in which it is a foot node in stead of a substitution node.

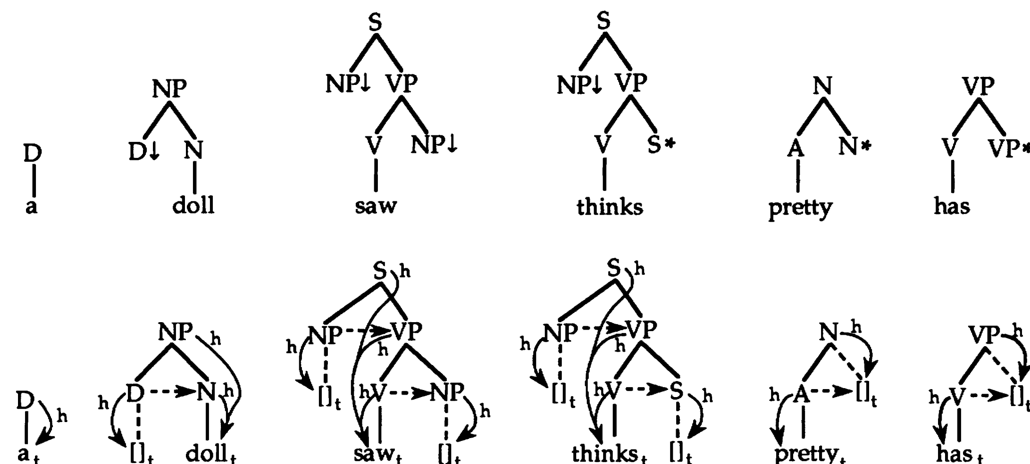


Figure 31: The top row shows LTAG elementary trees and the bottom row shows SUG structures which can be used to express the same grammatical information in an SUG grammar. An arrow marks nodes at which substitution must take place and a star marks the foot node of auxiliary trees.

The interesting distinctions between LTAG and SUG come out in their different mechanisms for handling long distance dependencies. Figure 32 gives a set of LTAG elementary trees for 'rides'. The first tree is the case without movement, the next two are the two possible extractions for wh-questions, and the last two are the two possible extractions for relative clauses. The filler-gap relationships in the last four trees can be stretched an unbounded distance by adjoining auxiliary trees, such as that given for 'thinks' in figure 31, at the lower S. If we take seriously the idea that the anchor of a tree is the source of the information in that tree, then the trees in figure 32 imply that the word 'rides' is ambiguous between the five extraction possibilities. It seems more natural to say that in each case the verb is the same, but the presence of a wh-word introduces the information about the long distance dependency. This analysis is easily expressed in SUG, as demonstrated in figure 33. The long distance dependency is still expressed locally, but in the structure for 'who' rather than in the structure for 'rides'[27].

[27]In a strictly lexicalized grammar this distribution of grammatical information runs into problems with

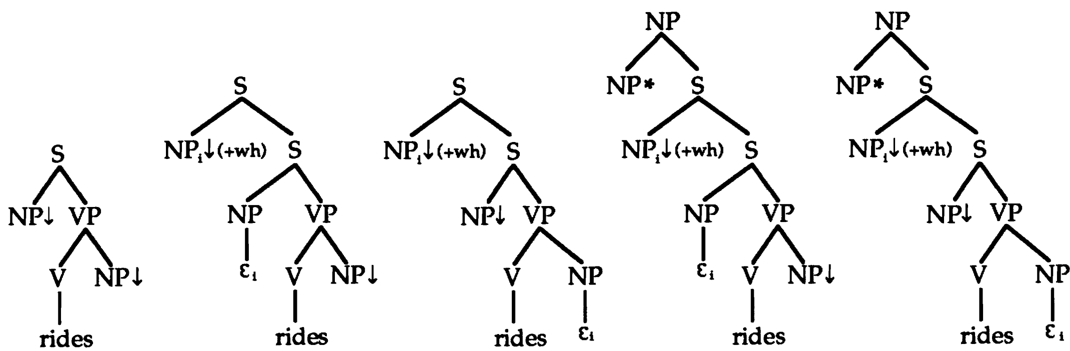


Figure 32: Five LTAG elementary trees for the word 'rides'. They are all necessary in order to express all the extraction possibilities.

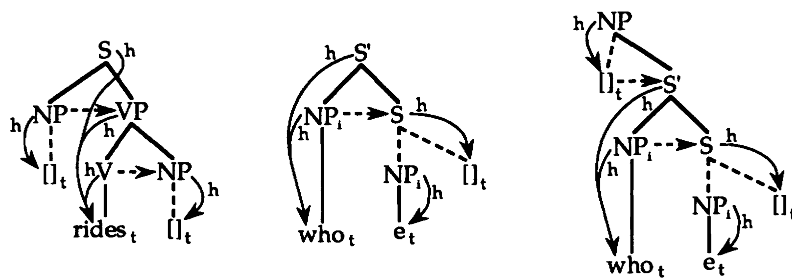


Figure 33: One SUG structure for 'rides' and two for 'who' which allow all the extraction possibilities expressed in figure 32.

Although the SUG analysis given here for long distance extraction more closely follows the intuition that the anchor should be the source of the information in its structure, as it is it doesn't express the constraints on extraction which are implied by the LTAG analysis. Because in the LTAG analysis a dependency can only stretch across structure for which there is an auxiliary tree, the possible extractions can be limited via restrictions on the possible auxiliary trees. As discussed in [Kroch, 1989], this approach allows restrictions on long distance extraction to fall out of purely local constraints on possible elementary trees. As it is the SUG analysis does not constrain the extraction possibilities at all, except via the dominance relation. However, the analysis in section 3.2 for expressing the bounding convention of LFG in SUG gives a technique for enforcing some restrictions with features. In particular, two features could be used, one which makes an unbounded coreference chain through the nodes which are foot nodes in LTAG trees, and one which establishes local domains along this chain. The trace would have to equate within one of these local domains along the unbounded chain from its dominating node. Such a system would restrict long distance dependencies in a similar way to LTAG, and have these restrictions fall out of purely local constraints on structures[28].

3.7 Combinatory Categorial Grammar

All the investigations discussed so far use a fairly traditional view of phrase structure, which is based primarily on semantic considerations. Combinatory Categorial Grammar (CCG, [Steedman, 1987]), on the other hand, makes no use of such structure. CCG advocates a more flexible form of constituent structure motivated by coordination and extraction phenomena. For example, in the sentence "Barbie pushed and Ken rode the tonka", "Barbie pushed" and "Ken rode" are each constituents, since they are coordinated, while in the sentence "Barbie pushed the tonka and surprised Ken", "pushed the tonka" and "surprised Ken" are each constituents. This view of constituency has allowed CCG to treat many such examples of "nonconstituent coordination" as simple constituent coordination.

reduced relatives, where there is no wh- word to introduce the modification and filler relationships. This would require another structure for 'rides' which carries the needed information, but it would still only be one such structure.

[28] Such use of features to restrict possible long distance dependencies may seem like a hack, but it could also be viewed as simply a declarative manifestation of a particular gap filling strategy in the natural language parser. I personally believe that restrictions on long distance dependencies are best investigated in a procedural framework.

This section will discuss how some of the insights gained from CCG's perspective on natural language phenomena are embodied in SUG. In particular, it will look at coordination phenomena. The central idea is that many SUG structures can be assigned types analogous to CCG categories. This permits the characterization of coordination phenomena given in CCG to be adopted for SUG. The constituent structures espoused by CCG are thus captured in the derivation structures of SUG, while still preserving conventional constituent structure in SUG's explicit representation of phrase structures.

Unfortunately, the types used as CCG categories are not sufficiently expressive to be used as types for SUG structures. To extend CCG's categories in a principled way it is necessary to return to their source, Categorial Grammar (CG). In the first section below a calculus is presented which adds a few features to Lambek's calculus for CG ([Lambek, 1961]), thus defining a system of types which is appropriate for typing SUG structures. These types are then used as categories in a system analogous to CCG. The resulting CCG–like system is equivalent to a large subset of SUG grammars, with an additional derivational constraint. Finally, some examples of coordination phenomena are given which demonstrate how CCG's theory of coordination can be applied to this subset of SUG via the new category system.

Applying CCG's notion of functional type to SUG structures has more significance than simply providing a theory of coordination for SUG. CCG's approach to language is very different from SUG's, but they both seem to reflect important characteristics of language. By crossing the two formalisms we can find a single representation which manifests the desirable characteristics of both formalisms. This process can also give us a better understanding of each formalism and their relation to each other. The approach taken here in combining the two formalisms is to first expand CCG until it can perspicuously express the linguistic information which has been found useful in SUG, then find an intersection of this expanded version of CCG and SUG. The formalism defined by this intersection will have two characterizations, one from the CCG side and one from the SUG side. Thus we can use the same formalism for investigating those characteristics of language which SUG reflects and for investigating those characteristics of language which CCG reflects, but still use the notation appropriate for the particular characteristic.

3.7.1 Categorial Grammar with Token Identity and Partiality

CCG is based on a system of types called Categorial Grammar. In [Lambek, 1961] Lambek defines a calculus (Lambek Calculus) for deducing equivalences between these syntactic

types. Many of the combination rules CCG uses are theorems of this calculus. This section adds some independently desirable features to CG syntactic types and modifies Lambek Calculus to reflect these changes. In particular, the new types include the ability to name and refer to specific tokens of categories, and the ability to underspecify the labels of basic categories and the ordering constraints between subcategories. The resulting system of types can be used to type SUG structures. In the following section theorems from this new calculus are used as combination rules in a system analogous to CCG which differs only minorly from a large and interesting subset of SUG grammars.

3.7.1.1 Lambek Calculus

The syntactic types of CG are either basic categories, such as NP or VP, or of the form (X/Y), (X\Y), or (X·Y), where X and Y are syntactic types. In this notation X/Y stands for "I would be an X if only I could combine with a Y to my right", and X\Y is the same except the Y is expected on the left. Thus, for example, the categories NP and S\NP can combine to form S. The category X·Y is the concatenation of X and Y.

Lambek Calculus uses one axiom and a set of inference rules to deduce subtype relationships. The sequents are of the form $\Delta \rightarrow$ X, where Δ is a sequence of types and X is a single type. This sequent means that a sequence of things with the types specified in Δ and in the order specified in Δ are also of the type X. The one axiom of the system is X $\rightarrow$ X, which expresses the trivial equivalence of identical types. The inference rules are as follows. The sequents above the line are the antecedents of the rule and the sequent below is the conclusion. For example, the /L rule should be read "if Δ is of type Y and Γ,X,Λ is of type Z, then Γ,X/Y,Δ,Λ is of type Z". The Cut rule is not actually needed, since adding it does not change the power of the system. The fact that this cut elimination theorem holds is important because it shows that this set of rules "make sense". What the Cut rule says is that, if Δ is of type X then, for any sequent you can prove with X on the left side, that sequent with Δ substituted for X is also a theorem. In other words, if Δ is of type X then anything which X can do, Δ can do. The fact that this is true even without having it explicitly stated in the Cut rule is why we can interpret this calculus as proving subtype relationships.

As an example of Lambek Calculus, the following is a proof of the composition rule X/Y Y/Z $\rightarrow$ X/Z.

$$/L: \frac{\Delta \rightarrow Y \quad \Gamma,X,\Lambda \rightarrow Z}{\Gamma,X/Y,\Delta,\Lambda \rightarrow Z}$$

$$\backslash L: \frac{\Delta \rightarrow Y \quad \Gamma,X,\Lambda \rightarrow Z}{\Gamma,\Delta,X\backslash Y,\Lambda \rightarrow Z}$$

$$/R: \frac{\Delta,Y \rightarrow X}{\Delta \rightarrow X/Y}$$

$$\backslash R: \frac{Y,\Delta \rightarrow X}{\Delta \rightarrow X\backslash Y}$$

$$\cdot L: \frac{\Delta,X,Y,\Lambda \rightarrow Z}{\Delta,X\cdot Y,\Lambda \rightarrow Z}$$

$$\cdot R: \frac{\Delta \rightarrow X \quad \Lambda \rightarrow Y}{\Delta,\Lambda \rightarrow X\cdot Y}$$

$$\text{Cut}: \frac{\Delta \rightarrow X \quad \Gamma,X,\Lambda \rightarrow Y}{\Gamma,\Delta,\Lambda \rightarrow Y}$$

$$\cfrac{\cfrac{Z \rightarrow Z \quad \cfrac{Y \rightarrow Y \quad X \rightarrow X}{X/Y,\ Y \rightarrow X}\ /L}{X/Y,\ Y/Z,\ Z \rightarrow X}\ /L}{X/Y,\ Y/Z \rightarrow X/Z}\ /R$$

3.7.1.2 Adding Token Identity

In Lambek Calculus, an instance of a category is described solely in terms of its type. There are no mechanisms for naming and referring to particular tokens of categories. For example, the category NP/NP restricts both the argument and the result of the category to be of type NP, but there is no way to say that these two NP's must be the same category token. This lack of expressive power carries over to CCG, where it prevents some necessary distinctions from being made. In this section I will present these linguistic motivations, then discuss how Lambek Calculus can be extended to allow identity between tokens of categories to be expressed and enforced. I will delay discussing the details of the resulting calculus until section 3.7.1.4.

Linguistic Motivations

The first major advantages of being able to refer to tokens of categories is the ability to distinguish between some categories which are indistinguishable in Lambek's categories. In Lambek's categories, one way of type raising an NP produces the category S/(S\NP). If an S/S is composed with this category, the result is also the category S/(S\NP). Thus a sentence needing a sentential complement, such as "Barbie said that" (S/S), can be combined with a type raised subject NP, such as "Ken" (S/(S\NP)), to form "Barbie said that Ken" (S/(S\NP)), and this string will be of the same type as "Ken" (S/(S\NP)).

The fact that these two strings can not be distinguished on the basis of their categories is a problem for CCG, because CCG relies on the equality of categories as its criteria for what can be coordinated. It is not possible to coordinate "Ken" with "Barbie said that Ken". If we add to the descriptions of these categories the ability to designate which uses of the category S actually describe the same token, then these categories are distinguishable. The type raised NP is now $S_i/(S_i\backslash NP)$ and the sentence looking for a verb phrase is now $S_i/(S_j\backslash NP)$, where equal subscripts designate token identity. Thus the added expressive power of being able to refer to tokens of categories prevents these two very different strings from having the same categories, and thus prevents CCG's rule for coordination from making the incorrect prediction that they should be coordinatable[29].

The second major advantage of being able to refer to tokens of categories is the ability to identify intermediate results within the category for a single word. For example, 'almost' can modify PP's but it would not be sufficient to give 'almost' the category (N\N)/(N\N), since this category would also allow 'almost' to modify other postnominal modifiers such as "who ate the cheese steak". To remedy this we can give prepositions a category such as $((N\backslash N)/PP_i)\cdot(PP_i/NP)$, which expresses the existence of the PP even though it is neither an argument nor a result for the category as a whole. With this analysis and the flexibility in ordering constraints proposed below, 'almost' can have the category PP_j/PP_j. Using this technique token identity provides a mechanism for incorporating many of the advantages of conventional phrase structure into a system with CCG derivation structures, as will be

[29] Mark Steedman (personal communication) has pointed out that the distinction I am making here already exists in the semantic structure which accompanies syntactic types in CCG. In the type raised category (S/(S\NP)), the two S's necessarily have the same semantic interpretation, while this is not true in the category for "Barbie said that Ken". However, allowing syntactic operations to be contingent on information available only in the semantic interpretation would be a radical departure from the system described in [Steedman, 1987]. The proposal given here can in part be viewed as characterizing the syntactic import of this semantically based generalization.

demonstrated below.

Extending Lambek Calculus

In order to be able to refer to particular tokens of categories, the calculus to be described below uses variables which range over categories. The categories which these variables name are specified in formulae. For example, $S_i/(S_i\backslash NP)$ can be expressed as w where *res(w)=x* ∧ *arg(w)=z* ∧ *dir(w)=rgt* ∧ *res(z)=x* ∧ *arg(z)=y* ∧ *dir(z)=lft* ∧ *cat(x)=S* ∧ *cat(y)=NP*. For convenience I will use the more readable form *w:(x:S/z:(x\y:NP))*. Since the result of the category and the result of the argument of the category are named with the same variable, the token identity of these two positions has been expressed.

Given the ability to express token identity between categories, the calculus has to be changed to enforce these constraints. The problem is that the places where a category is mentioned may be far apart in a sequent, and the places in a proof tree where a category is involved in a proof step may also be far apart. To solve this problem the proof needs to construct a derivation history of each category and pass that history through the proof. In this way each proof step can be contingent on being compatible with the previous steps of the proof.

The derivation history constructed by a proof needs to record both what categories were produced from what other categories, and the necessary ordering of these categories. For example, if the /L rule combines *z:(x/y)* with *w* to make *x*, then the history needs to record that *y*=*w*, that *z* precedes *w*, and that *z* and *w* were used to produce *x*. Also, the system has to include rules which propagate ordering constraints through the derivation history and prevent incompatible information from existing in a proof's history information. The details of such a system will be presented after discussing adding partiality to the categories.

3.7.1.3 Adding Partiality

The categories of Lambek Calculus do not allow for much partial information. First, the basic categories are atomic symbols, such as S or NP. There is no way to partially specify these categories. The basic categories should be specified as feature structures, so as to allow their partial specification. Second, the only way to express ordering constraints is via the directionality of slashes. This mechanism only allows siblings in the derivational history to be ordered, and all siblings must be ordered. This does not allow sufficient flexibility for expressing ordering constraints. Thus a more flexible mechanism for expressing ordering

constraints between categories needs to be added. This section will give some linguistic arguments for these additions, then discuss how they can be added to the calculus discussed in the previous section. The complete calculus will be presented in the next section.

Linguistic Motivations

Many people have argued for the extensive use of partial information in linguistic formalisms ([Marcus *et al.*, 1983], [Shieber, 1986], etc.). Section 3.1.2 gave examples of when feature structures are important for underspecifying node labels. The atomic symbols used in most work on categorial grammar are often viewed as a simpler notation for a more complete feature structure representation. With this extension two categories match if they have the same function–argument structure and the corresponding basic categories from each category can unify.

The inadequacies of atomic basic categories are not as severe a limitation as the extremely limited ability to express ordering constraints. As an example of this, consider the problems CCG has with handling some long distance dependencies. When parsing the sentence "who does Barbie think poses", the word 'who' has either the category S′/(S/NP) or the category S′/(S\NP). However, the phrase "does Barbie think poses' has neither the category S/NP nor the category S\NP, since the NP it is missing is neither on its right nor on its left. Versions of CCG have been proposed to handle this problem, but it is difficult to prevent these versions from allowing too much flexibility in the positions of NP's. The approach taken here is that the category for 'who' should express the fact that the NP missing from its S argument can be internal to the S. To do this category specifications need to be able to leave categories unordered[30].

In [Joshi, 1987b], Joshi argues that not only do siblings sometimes need to be unordered, but that some languages require ordering constraints between nonsiblings. To provide these expressive abilities, the new calculus does not use directionality of slashes to represent ordering constraints. In stead, ordering constraints are expressed directly between tokens of categories. This allows any ordering constraints expressible within the domain of a lexical entry, which is comparable to that of the version of TAG presented in [Joshi, 1987b]. This system still has the ability to express any ordering constraints expressible using directional slashes.

[30]This is not the same as the notation (S|NP), which is simply an abbreviation for (S/NP ∨ S\NP).

Extending the Calculus

Given the changes to the calculus to allow reference to tokens of categories, it is easy to add the partial specification of basic categories and ordering constraints just discussed. Since categories are already specified using formulae to restrict the instantiation of variables, the methods used in section 2.1 to express feature structures in this form can be used. In a proof, the necessary unifications of basic category labels will occur as a side effect of expressing the token identity of these categories, since unification is done simply by equating variables. As with other constraints, the calculus must be restricted so that no inconsistent information can be created in the course of doing these equations.

Since the ordering constraints described above are specified directly between categories, rather than being expressed in the directionality of slashes, the inference rules do not need to mention ordering constraints. As in section 2.1, the ordering constraints can simply be expressed as predications over categories. I will use the same ordering relation used for SUG, namely linear precedence. In order to enforce these predications, the formulae of a sequent need to be interpreted with respect to a few axioms which propagate the ordering constraints down the derivation history and prevent incompatible ordering constraints. These axioms will be specified in the next section.

3.7.1.4 The New Calculus

The calculus presented here extends Lambek Calculus by adding the ability to refer to specific tokens of categories, allowing basic categories to be partially specified with feature structures, and relaxing the method of expressing ordering constraints to allow the specification of linear precedence constraints between arbitrary categories. First the sequents of this calculus are defined, then the axiom and rules are given. Finally several example proofs are given.

The Sequents

As discussed above, this calculus uses variables to refer to specific tokens of categories and formulae to constrain the instantiation of these variables. In accordance with this, sequents are of the following form, where S is a multiset of variables, x is a variable, and f_1 and f_2 are formulae which constrain the possible values for the variables. Ordering

$\langle S, f_1 \rangle \rightarrow \langle \mathrm{x}, \mathrm{f}_2 \rangle$

constraints between the categories in S are expressed in f_1. I will represent multisets using square brackets, as in $[x_1, \ldots, x_n]$.

In a sequent, the variables designate which tokens of categories are under consideration, and the formulae specify all other information. The formulae determine the categories of the tokens, the ordering constraints on the tokens, and the produced-from relations between tokens. These formulae will be defined after first specifying the terms of these formulae.

The terms of the formulae are typed. All terms are of type category ($\mathcal{C}$). The categories are divided into concatenation categories ($\mathcal{C}_c$) and function categories ($\mathcal{C}_f$). The function categories are also divided into those which are zeroth order ($\mathcal{C}_{f^0}$), and those which are nonzeroth order ($\mathcal{C}_{f^+}$). The zeroth order function categories are basic categories, and thus are feature structures.

Given a set A of atoms, a set L of labels, and a set V of variables, the formulae terms are defined as the following basic terms closed under the subsequent term constructors.

basic terms: $a \in A: \mathcal{C}_{f^0}$, $\quad x \in V: \mathcal{C}$

constructors: *frst:* $\mathcal{C}_c \rightarrow \mathcal{C}$, $\quad$ *scnd:* $\mathcal{C}_c \rightarrow \mathcal{C}$, $\quad$ *res:* $\mathcal{C}_{f^+} \rightarrow \mathcal{C}_f$, $\quad$ *arg:* $\mathcal{C}_{f^+} \rightarrow \mathcal{C}_f$, $\quad$ $l \in L: \mathcal{C}_{f^0} \rightarrow \mathcal{C}_{f^0}$

The atoms are the basic feature structures. The constructors frst and scnd map concatenation categories[31] to their first and second components. For function categories, res and arg specify their result and argument. Nonatomic feature structures are specified using labels as functions which map feature structures to feature structures, as in section 2.1.

The formulae are as follows, where $t_1, t_2 \in \mathcal{C}$ and f_1 and f_2 are formulae.

$$t_1 \approx t_2, \quad t_1 \prec t_2, \quad t_1 \leftarrow t_2, \quad f_1 \wedge f_2, \quad \top, \quad \bot$$

The symbol $\approx$ is the same slight modification of equality used for SUG, which is discussed in section 2.1.3. Ordering constraints are specified using $\prec$ for linear precedence. The produced-from relation is specified using $\leftarrow$. The only connective allowed in formulae is conjunction. The symbol $\top$ is true and $\bot$ is false.

The intended meaning of the above terms and predicates dictate that formulae have certain properties. These properties are enforced by the following axioms.

[31] The name "concatenation category" is not entirely accurate, since the order of the two components is not important in this calculus. As with the directionality of slashes, ordering constraints between concatenated categories need to be expressed independently.

$$x \leftarrow x$$

$$x \leftarrow y \wedge y \leftarrow z \Rightarrow x \leftarrow z,$$

$$x \prec y \wedge y \prec z \Rightarrow x \prec z,$$

$$x \prec y \wedge x \leftarrow z \wedge y \leftarrow w \Rightarrow z \prec w$$

$$\neg(x \prec x)$$

In addition to these axioms, feature structures are constrained by the first four axioms in section 2.1.3, which were taken from [Johnson, 1990]. The above axioms are simply those from section 2.1.3 which are concerned with dominance and precedence, where dominance is manifested here as $\leftarrow$.

The Axioms and Rules

This calculus is designed to have certain properties which guarantee that it has the desired behavior. First, in a proof, information is passed up through the left sides of sequents to an axiom, where it is checked for consistency, and down the right sides of sequents until it is either passed back up another portion of the proof or expressed in the theorem. The ·R rule also has to check for consistency, since it combines two formulas on the right side. Second, the only information which can be added to the formulae as they are passed through the proof is that information required by the rules and axioms. Arbitrary information can not be added, even though such information would only weaken the resulting theorem. Third, the category on the right side of a sequent needs to be produced from all the categories on the left side, in order to enforce ordering constraints. Thus the rules and axioms are designed so that if "$\langle S, f_1\rangle \rightarrow \langle x, f_2\rangle$" is a theorem, then f_2 entails f_1 and for all $y \in S$, $x \leftarrow y$.

The axioms are as follows.

axioms: $\langle [x], f_1\rangle \rightarrow \langle y, f_1 \wedge x \approx y\rangle$
where $(f_1 \wedge x \approx y)$ is satisfiable.

The unifiability of basic categories and the consistency of ordering constraints are guaranteed by requiring that the right side formula is satisfiable.

The inference rules are designed to enforce the information passing requirements given above. For convenience I will use $z \equiv x/y$ for $(res(z) \approx x \wedge arg(z) \approx y)$ and $z \equiv x \cdot y$ for $(frst(z) \approx x \wedge scnd(z) \approx y)$. Remember that the slash in "x/y" is now nondirectional, as is the connective in "$x \cdot y$".

$$/L:\ \frac{\langle S_1,\ f_1\rangle \rightarrow \langle y,\ f_2\rangle \quad \langle S_2{+}[x],\ f_2 \wedge z{\equiv}x/y \wedge x{\leftarrow}z \wedge x{\leftarrow}y\rangle \rightarrow \langle w,\ f_3\rangle}{\langle S_1{+}S_2{+}[z],\ f_1\rangle \rightarrow \langle w,\ f_3\rangle}$$

$$/R:\ \frac{\langle S{+}[y],\ f_1 \wedge z{\equiv}x/y \wedge z{\leftarrow}x\rangle \rightarrow \langle x,\ f_2\rangle}{\langle S,\ f_1\rangle \rightarrow \langle z,\ f_2\rangle}$$

$$\cdot L:\ \frac{\langle S{+}[x,y],\ f_1 \wedge z{\equiv}x{\cdot}y \wedge z{\leftarrow}x \wedge z{\leftarrow}y\rangle \rightarrow \langle w,\ f_2\rangle}{\langle S{+}[z],\ f_1\rangle \rightarrow \langle w,\ f_2\rangle}$$

$$\cdot R:\ \frac{\langle S_1,\ \ f_1 \wedge z{\equiv}x{\cdot}y \wedge z{\leftarrow}x\rangle \rightarrow \langle x,\ \ f_2\rangle \quad \langle S_2,\ \ f_1 \wedge z{\equiv}x{\cdot}y \wedge z{\leftarrow}y\rangle \rightarrow \langle y,\ \ f_3\rangle}{\langle S_1{+}S_2,\ \ f_1\rangle \rightarrow \langle z,\ \ f_2 \wedge f_3\rangle}$$

where $(f_2 \wedge f_3)$ is satisfiable.

$$Cut:\ \frac{\langle S_1,\ \ f_1\rangle \rightarrow \langle x,\ \ f_2\rangle \quad \langle S_2{+}[x],\ \ f_2\rangle \rightarrow \langle y,\ \ f_3\rangle}{\langle S_1{+}S_2,\ \ f_1\rangle \rightarrow \langle y,\ \ f_3\rangle}$$

Including the Cut rule is redundant, as it is in Lambek Calculus. As discussed above for Lambek Calculus, this fact indicates that this proof system can be interpreted as deriving subtype relationships.

Example Proofs

To demonstrate this new calculus, below are given several proof trees for interesting theorems. These theorems will be used below in defining an expanded version of CCG.

The proof of function application:

$$\frac{\langle [y_2],\ u{\equiv}x/y_1\rangle \rightarrow \left\langle y_1,\ \begin{matrix} u{\equiv}x/y_1 \\ \wedge\ y_1{\approx}y_2 \end{matrix}\right\rangle \quad \left\langle [x],\ \begin{matrix} u{\equiv}x/y_1 \wedge y_1{\approx}y_2 \\ \wedge\ x{\leftarrow}u \wedge x{\leftarrow}y_1 \end{matrix}\right\rangle \rightarrow \left\langle x,\ \begin{matrix} u{\equiv}x/y_1 \wedge y_1{\approx}y_2 \\ \wedge\ x{\leftarrow}u \wedge x{\leftarrow}y_1 \end{matrix}\right\rangle}{\langle [u,y_2],\ u{\equiv}x/y_1\rangle \rightarrow \langle x,\ u{\equiv}x/y_1 \wedge y_1{\approx}y_2 \wedge x{\leftarrow}u \wedge x{\leftarrow}y_1\rangle}\ /\mathrm{L}$$

The proof of function composition with one argument:

$$
\cfrac{
\langle [z],\ F_1\rangle \rightarrow \langle z,\ F_1\rangle
\qquad
\cfrac{
\left\langle [y_2], F_2 \right\rangle \rightarrow \left\langle y_1, \begin{array}{l} y_1 \approx y_2 \\ \wedge\ F_2 \end{array} \right\rangle
\quad
\left\langle [x], \begin{array}{l} x \leftarrow u \\ \wedge\ x \leftarrow y_1 \\ \wedge\ y_1 \approx y_2 \\ \wedge\ F_2 \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow u \\ \wedge\ x \leftarrow y_1 \\ \wedge\ y_1 \approx y_2 \\ \wedge\ F_2 \end{array} \right\rangle
}{
\left\langle [u,y_2], \begin{array}{l} (y_2 \leftarrow v \wedge y_2 \leftarrow z \wedge F_1) \\ = F_2 \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow u \wedge x \leftarrow y_1 \\ \wedge\ y_1 \approx y_2 \wedge F_2 \end{array} \right\rangle
}\ /\mathrm{L}
}{
\cfrac{
\left\langle [u,v,z], \begin{array}{l} (w \equiv x/z \wedge w \leftarrow x \\ \wedge\ u \equiv x/y_1 \wedge v \equiv y_2/z) \\ = F_1 \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow u \wedge x \leftarrow y_1 \\ \wedge\ y_1 \approx y_2 \wedge F_2 \end{array} \right\rangle
}{
\langle [u,v],\ u \equiv x/y_1 \wedge v \equiv y_2/z\rangle \rightarrow \langle w,\ x \leftarrow u \wedge x \leftarrow y_1 \wedge y_1 \approx y_2 \wedge F_2\rangle
}\ /\mathrm{R}
}\ /\mathrm{L}
$$

The proof of the insignificance of argument order for the outermost two arguments:

$$
\cfrac{
\langle [z],\ F_1\rangle \rightarrow \langle z,\ F_1\rangle
\qquad
\cfrac{
\left\langle [y], \begin{array}{l} v \leftarrow u \\ \wedge\ v \leftarrow z \\ \wedge\ F_1 \end{array} \right\rangle \rightarrow \left\langle y, \begin{array}{l} v \leftarrow u \\ \wedge\ v \leftarrow z \\ \wedge\ F_1 \end{array} \right\rangle
\quad
\left\langle [x], \begin{array}{l} x \leftarrow v \\ \wedge\ x \leftarrow y \\ \wedge\ v \leftarrow u \\ \wedge\ v \leftarrow z \\ \wedge\ F_1 \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow v \\ \wedge\ x \leftarrow y \\ \wedge\ v \leftarrow u \\ \wedge\ v \leftarrow z \\ \wedge\ F_1 \end{array} \right\rangle
}{
\langle [v,y],\ v \leftarrow u \wedge v \leftarrow z \wedge F_1\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow v \wedge x \leftarrow y \\ \wedge\ v \leftarrow u \wedge v \leftarrow z \wedge F_1 \end{array} \right\rangle
}\ /\mathrm{L}
}{
\cfrac{
\cfrac{
\left\langle [u,y,z], \begin{array}{l} (t \equiv x/z \wedge t \leftarrow x \wedge s \equiv t/y \\ \wedge\ s \leftarrow t \wedge u \equiv v/z \wedge v \equiv x/y) \\ = F_1 \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} x \leftarrow v \wedge x \leftarrow y \\ \wedge\ v \leftarrow u \wedge v \leftarrow z \wedge F_1 \end{array} \right\rangle
}{
\left\langle [u,y], \begin{array}{l} s \equiv t/y \wedge s \leftarrow t \\ \wedge\ u \equiv v/z \wedge v \equiv x/y \end{array} \right\rangle \rightarrow \left\langle t, \begin{array}{l} x \leftarrow v \wedge x \leftarrow y \\ \wedge\ v \leftarrow u \wedge v \leftarrow z \wedge F_1 \end{array} \right\rangle
}\ /\mathrm{R}
}{
\langle [u],\ u \equiv v/z \wedge v \equiv x/y\rangle \rightarrow \left\langle s, \begin{array}{l} x \leftarrow v \wedge x \leftarrow y \\ \wedge\ v \leftarrow u \wedge v \leftarrow z \wedge F_1 \end{array} \right\rangle
}\ /\mathrm{R}
}\ /\mathrm{L}
$$

The proof of type raising:

$$
\cfrac{
\cfrac{
\left\langle [x], \begin{array}{l} w \equiv y/v \\ \wedge\ w \leftarrow y \end{array} \right\rangle \rightarrow \left\langle x, \begin{array}{l} w \equiv y/v \\ \wedge\ w \leftarrow y \end{array} \right\rangle
\quad
\left\langle [y], \begin{array}{l} v \equiv y/x \\ \wedge\ y \leftarrow v \wedge y \leftarrow x \\ \wedge\ w \equiv y/v \wedge w \leftarrow y \end{array} \right\rangle \rightarrow \left\langle y, \begin{array}{l} v \equiv y/x \\ \wedge\ y \leftarrow v \wedge y \leftarrow x \\ \wedge\ w \equiv y/v \wedge w \leftarrow y \end{array} \right\rangle
}{
\langle [x,v],\ w \equiv y/v \wedge w \leftarrow y\rangle \rightarrow \langle y,\ v \equiv y/x \wedge y \leftarrow v \wedge y \leftarrow x \wedge w \equiv y/v \wedge w \leftarrow y\rangle
}\ /\mathrm{L}
}{
\langle [x],\quad \rangle \rightarrow \langle w,\ v \equiv y/x \wedge y \leftarrow v \wedge y \leftarrow x \wedge w \equiv y/v \wedge w \leftarrow y\rangle
}\ /\mathrm{R}
$$

3.7.2 CCG with Structure Categories

Now that we have a calculus which adds to Lambek Calculus the ability to specify token identity, partial basic categories, and partial ordering constraints, we can make these additions to Combinatory Categorial Grammar. CCG uses the categories from Lambek Calculus and combines them using a few rules whose application can be restricted. For the most part, these combination rules are theorems of Lambek Calculus. In particular, CCG uses function application and function composition. One exception is the use of non-order-preserving function composition. For example the rule $X/Y\ Y\backslash Z \rightarrow X\backslash Z$ is sometimes used in CCG but it is not a theorem of Lambek Calculus. These operations do not seem to be necessary with the ability to flexibly specify ordering constraints, so I will not include them in the version of CCG to be defined here. The other example is used for handling parasitic gaps, which will not be discussed here. With these provision, all the components of CCG are taken from Lambek Calculus. Thus we can define a new version of CCG with the desired additions simply by using the categories and theorems of the calculus presented in the previous section rather than Lambek Calculus. Since this new calculus is an extension of Lambek Calculus, the resulting system will still be able to express all the categories expressible in CCG.

In addition to the above changes, the version of CCG defined here has several restrictions which only allow categories which behave like SUG structures. The most significant of these is that categories can not be greater than second order. Other restrictions prevent certain identities between categories. None of these restrictions interfere with the linguistic applications of this system. The resulting version of CCG will be called Structural Combinatory Categorial Grammar (SCCG).

This section defines SCCG in more detail and gives a mapping from SCCG grammars to equivalent grammars in a slightly restricted version of SUG. The restriction manifests a constraint in SCCG on possible long distance dependencies. This mapping will be used below to show how the theory of coordination in CCG can be applied in SUG.

3.7.2.1 Structural Combinatory Categorial Grammar

Like CCG, an SCCG grammar is lexical. The categories for SCCG are those defined for the extension of Lambek Calculus given above. In order to maintain some of the advantages discussed in section 3.7.1, each lexical entry contains a set of categories, rather than

a single category, plus an anchor category for the entry's word. An anchor category determines the position of its word. In a derivation, the possible orderings of the words is exactly the possible orderings of the anchor categories for the words, as determined by the ordering constraints between these anchor categories. The possible grammar entries will be constrained, as outlined above, after the derivations of SCCG are defined.

In CCG the main combination operations are function application and function composition. SCCG uses versions of these operations provable in the above extension of Lambek Calculus. Proofs for some of these rules were given above. In addition to these rules, SCCG allows the order of arguments to be switched. This is in keeping with the use of linear precedence statements to enforce ordering constraints. One case of the rules which switch arguments was also proven above.

Since the ability to rearrange the order of arguments makes all orderings equivalent, I will use a notation for categories which does not represent this superfluous information[32]. In particular, categories will be represented as a result, which must be a basic category, and a multiset of argument categories. For example, the category w where $res(w) \approx v \wedge arg(w) \approx z \wedge res(v) \approx x \wedge arg(v) \approx y$ will now be w where $res(w) \approx x \wedge arg(w) \approx [y,z]$. For readability I will write this as $w \equiv x/[y,z]$. The notation $y \equiv x/[]$ will be interpreted as $y \approx x$. In SCCG derivations it will often be convenient to specify this category information when the category is mentioned. For this I will use the notation $w{:}(x{:}S/[y{:}NP,\ z{:}VP])$ for w where $w \equiv x/[y,z] \wedge cat(x) \approx S \wedge cat(y) \approx NP \wedge cat(z) \approx VP$. In such cases the derivation history and ordering information will be specified separately. For example, $x \leftarrow w \wedge x \leftarrow y \wedge x \leftarrow z \wedge y \prec z$ will be written $\left\langle \begin{array}{c} x \text{———} z \\ w \;\; y \end{array} \wedge y \prec z \right\rangle$. In derivations, the later information will be left out where it is not important. The anchor category for each word will be written in bold face.

The change in notation just introduced allows the rules of SCCG to be specified in one rule schema, but I will introduce it through a series of increasingly more general rules. The rule schema below is for function application. As is implied by the notation, the order of the arguments $[z_1,\ldots,z_n,\ y]$ is not important. Because the rules which change the ordering of arguments are particular to the number of arguments involved, this schema is actually short hand for an infinite number of rules, since there can be an arbitrary number of categories in $r_1,\ldots,r_n$ which y might need to be moved over.

[32]This change eliminates from the notation categories such as the x/y in $(x/y)/z$. This is only significant if there are linear precedence relations specified on such intermediate categories. The restrictions on possible grammar entries discussed below ensure that no such constraints are specified.

$$\frac{u{:}(x/[r_1,\ldots,r_n,\ y])\ \langle f_1\rangle \qquad\qquad y'\langle f_2\rangle}{w{:}(x/[r_1,\ \ldots,\ r_n])\ \left\langle f_1 \wedge f_2 \wedge y\approx y' \wedge \begin{array}{l} x\!-\!y \\ \quad\searrow u \end{array} \right\rangle}$$

The rule for function composition is the same except y' is replaced by $y'//[s_1,\ldots,s_m]$ and $s_1,\ldots,s_m$ are passed on as arguments in the resulting category. The resulting schema is shown below. Since in this notation $y'//[]=y'$, the schema shown above is a special case of this one, except that now y', and thus y, must be basic categories.

$$\frac{u{:}(x/[r_1,\ldots,r_n,\ y])\ \langle f_1\rangle \qquad v{:}(y'//[s_1,\ldots,s_m])\ \langle f_2\rangle}{w{:}(x/[r_1,\ldots,r_n,\ s_1,\ldots,s_m])\ \left\langle \begin{array}{l} f_1 \wedge f_2 \wedge y\approx y' \\ \wedge\quad w\!-\!x\!-\!y\!-\!v \\ \qquad\quad u \quad s_1\ldots s_m \end{array} \right\rangle}$$

To allow the argument y to be nonbasic we need to express its possible arguments explicitly. Thus y should be $y/[z_1,\ldots,z_p]$ and v should be $(y'//[z'_1,\ldots,z'_p])/[s_1,\ldots,s_m]$. In the current notation v needs to be written $y'//[z'_1,\ldots,z'_p,\ s_1,\ldots,s_m]$. These changes result in the rule schema given below. Again, since $y/[]=y$, the above rule schema is a special case of this one. This schema represents the combination rules of SCCG. Note that t and v are not equated, only their subcategories are. Since SCCG categories are never greater than second order, all the categories which are equated are basic categories.

$$\frac{u{:}(x/[r_1,\ldots,r_n,\ t{:}(y/[z_1,\ldots,z_p])])\ \langle f_1\rangle \quad v{:}(y'//[z'_1,\ldots,z'_p,\ s_1,\ldots,s_m])\ \langle f_2\rangle}{w{:}(x/[r_1,\ldots,r_n,\ s_1,\ldots,s_m])\ \left\langle \begin{array}{l} f_1 \wedge f_2 \wedge y\approx y' \wedge z_1\approx z'_1 \wedge \ldots \wedge z_p\approx z'_p \\ \wedge\quad w\!-\!x\!-\!t\!-\!y\!-\!v \\ \qquad\quad u \quad z_1\ldots z_p \quad s_1\ldots s_m \end{array} \right\rangle}$$

A SCCG derivation proceeds by using the above rule schema to combine categories from the chosen lexical entries for the words. When the categories have been reduced to a single basic category, the derivation is complete. Each combination of categories must not introduce inconsistent category information. An example of a derivation in SCCG is given below. The grammar entry for 'poses' has three categories in it in order to express the internal structure of the projection to S. This is represented with the concatenation symbols between the categories. To help show the structural nature of this system, category information is included in the produced–from tree. The ordering constraints imposed by the order of the words are introduced as the derivation proceeds.

The definition of SCCG given so far is further constrained with some restrictions on

Ken	poses	shamelessly
$\mathbf{x_1{:}NP}$	$y{:}S/[x_2{:}NP,\ z_1]\ \langle x_2 \prec z_1 \rangle \cdot \mathbf{w{:}V} \cdot z_1{:}VP/[w]$	$\mathbf{v{:}(z_2{:}VP/[z_2])}$

$$y{:}S/[z_1]\ \left\langle \begin{array}{l} x_2 \prec z_1 \wedge x_1 \prec w \\ \wedge\ x_1 \approx x_2\ \wedge \\ y{:}S \langle z_1{:}VP,\ x_1{:}NP \rangle \end{array} \right\rangle \qquad z_1{:}VP/[w{:}V]\ \left\langle \begin{array}{l} w \prec v \wedge z_1 \approx z_2 \\ \wedge\ z_1{:}VP \langle v{:}(z_1/[z_1]),\ w{:}V \rangle \end{array} \right\rangle$$

$$z_1{:}VP\ \left\langle w \prec v \wedge z_1 \approx z_2 \wedge z_1{:}VP \langle v{:}(z_1/[z_1]),\ w{:}V \rangle \right\rangle$$

$$y{:}S\ \left\langle \begin{array}{l} x_2 \prec z_1 \wedge x_1 \prec w \wedge w \prec v \wedge x_1 \approx x_2 \wedge z_1 \approx z_2 \\ \wedge\ y{:}S \langle z_1{:}VP \langle v{:}(z_1/[z_1]),\ w{:}V \rangle,\ x_1{:}NP \rangle \end{array} \right\rangle$$

possible grammar entries. The first of these is that no category can be greater than second order. This permits the category $x/[y/z]$ but not $x/[y/[z/[w]]]$. The only potential linguistic application for categories of greater than second order are modifiers of modifiers of modifiers. In CCG such a word would have a category like ((X/X)/(X/X))/((X/X)/(X/X)), which is third order. However the ability to specify internal structure in SCCG permits modifiers to introduce nodes which permit categories which modify them to be first order, as was done with PP's and 'almost' in section 3.7.1.2. Thus there seems to be no linguistic application for categories greater than second order in SCCG.

The second constraints on SCCG grammar entries is more complicated but seems to be necessary in order to give a structural interpretation to SCCG categories. The basic idea is that a given category will be introduced into a derivation by only one category and will be removed from the derivation by only one category. For example, the category x/y removes y from the derivation and introduces x. The category x/x does not introduce or remove any categories. The category $x/(y/z)$ introduces both x and z, and removes y[33]. To enforce this constraint, for every time a category is introduced that category must have an instance unique value for the feature *introduced*, and for every time a category is removed that category must have an instance unique value for the feature *removed*. This prevents both the initial specification of a category as being introduced or removed in more than one position, and the equation of two categories which are each introduced or removed in some category. This is the only use of instance unique values allowed in SCCG.

The definition of when a category is introduced and when it is removed uses three positions in a category. They are the result, the argument results, and the argument arguments.

[33]The reason $x/(y/z)$ must be treated as introducing z is that y/z removes z and $x/(y/z)$ combines with y/z to make x, which does not remove z, so z must have been introduced by $x/(y/z)$.

Given a category $x \equiv (z/[(w_1/[u_{11},\ldots,u_{1m}]),\ldots,(w_n/[u_{n1},\ldots,u_{np}])])$, z is the result, the w_i are the argument results, and the u_{ij} are the argument arguments. Remember that in this notation $x{=}(x/[])$, so even if a category has no arguments, it still has a result, namely itself. Since results are all basic categories and no categories are greater than second order, all categories in one of these positions are basic categories. A category x introduces a category y if y is mentioned as either a result or an argument argument, more times than y is mentioned as an argument result. A category x removes a category y if y is mentioned as an argument result more times than y is mentioned as either a result or an argument argument. If the difference between these two counts is greater than one, then y is introduced or removed that many times, but this will not occur with the categories allowed in SCCG.

The third constraint is very simple; no produced–from relations can be stated in SCCG grammar entries. This ensures that the produced–from relation in the result of an SCCG derivation will be exactly the history information for that derivation.

The fourth restriction on SCCG grammar entries requires that the feature structure label of a category in an argument argument position must be incompatible with that of its argument result category. In other words, if a category has an argument u and u has a result y and an argument z, then y and z must not be unifiable. This constraint may not be necessary, but it greatly simplifies the relationship between SCCG and SUG.

The last constraint restricts what categories can be mentioned in constraints in SCCG grammar entries. The only categories on which constraints can be stated are basic categories and categories specified as elements in grammar entries. For example, if $w{:}(v{:}(x{:}S/y{:}NP)/z{:}VP)$ is an element of a grammar entry, then w, x, y, and z can have linear precedence statement specified on them, but v cannot. This is partly to simplify the translation to SUG and partly because in the notation used for SCCG, in this example $w{:}(x/[y,z])$, such internal categories do not exist. This restriction means that only these categories can be mentioned in linear precedence relations, and only these categories can be the anchor category of a grammar entry. Since all values in feature structures must be of type C_{f^0}, node label constraints are already limited to only basic categories.

3.7.2.2 SCCG's Relation to SUG

With the above definition there is a fairly direct relationship between SCCG categories and a subset of SUG structures. As has been used repeatedly in previous sections in this chapter, SUG nodes can be forced to equate with other nodes using underspecified terminals and

features such as *uid* or *head.* Such nodes correspond to arguments in SCCG categories. SUG nodes which fill these underspecified terminals correspond to results in SCCG categories. The use of nonredundant dominance links in SUG structures correspond to the use of second order functions in SCCG, with one restriction to be discussed. Examples of these relationships are given in figure 34. This section will define these correspondences in more detail by giving a mapping from SCCG categories to weakly equivalent SUG structures, and a necessary restriction on the instantiation of dominance links in SUG. All the SCCG grammar's derivations are included in their equivalent SUG grammar's derivations. Because of these relationships SCCG can be used to show the connection between the analysis of coordination in CCG and equivalent analyses in SUG. This connection is the topic of the next section.

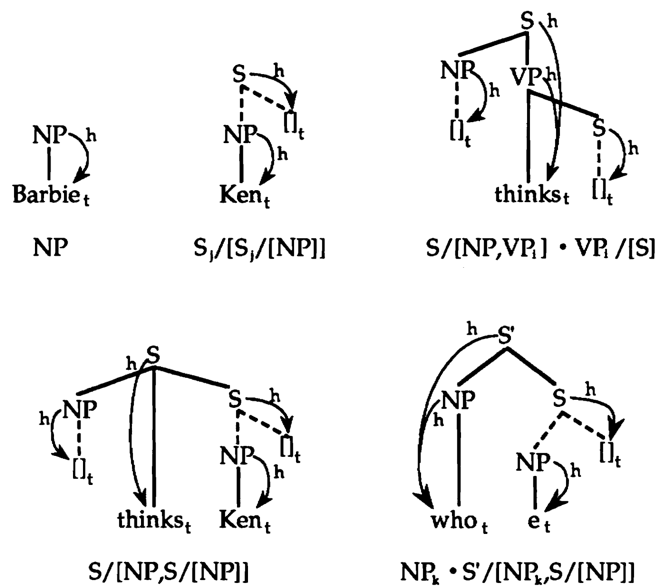


Figure 34: Some examples of SUG structures and the SCCG categories which are equivalent to them.

The categories of SCCG have a lot in common with SUG structures. SCCG categories have linear precedence relations and produced–from relations which behave exactly like linear precedence and dominance relations in SUG. The combination operation is SCCG results in the equation of categories, and the combination operation in SUG results in the equation of nodes. SCCG basic categories are feature structures, as are SUG nodes. When a SCCG derivation is done there must be a single category which is produced from all the

categories used in the derivation. Likewise, when an SUG derivation is done there must be a single node which dominates all the other nodes used in the derivation. The only aspect of SCCG categories which does not have a trivial correlate in SUG is the requirement that all arguments to categories must find other categories to equate with and all results except the final result must find arguments to equate with. In SUG these properties can be enforced using the two requirements on the completion of an SUG derivation: that all terminals must have their words specified and that all nodes except the root must have an immediate parent.

With the above outline of the correlation between SCCG categories and SUG structures it is fairly easy to define a mapping from SCCG categories to equivalent SUG structures. Let the SCCG category being translated be c. Also let h be a function which maps categories in c to nodes in the SUG translation of c. For all basic categories x in c, *h(x)* is a nonterminal. If c is not basic and is an anchor category for a word w, then *h(c)* is a terminal with w as its word and the h of the result of c is the immediate parent of *h(c)*. If c is or contains a basic anchor category x for a word w, then create a terminal a with w as its word and state *idom(h(x),a)* in the SUG structure. For all the feature structure information stated about a basic category x in c, state the same information about *h(x)*, except the *introduced* and *removed* feature information. Do the same for linear precedence information, ignoring constraints on categories whose h has not been defined. In this same way, translate produced–from constraints between these categories as dominance relations in the SUG structure[34].

As outlined above, the argument–result information in SCCG categories can be manifested in SUG structures using immediate dominance links and terminals. To describe this mapping I will make use of the definitions of the category's result, the category's argument results, the category's argument arguments, when a category introduces another category, and when it removes another category, which were all given in section 3.7.2.1. For all the basic categories x in the category being translated c, give *h(x)* a feature *head* which has as its value a terminal. This terminal has no word specified unless otherwise stated. If c removes a category x, then in the SUG structure, *h(x)* is immediately dominated by the h of the result of c, and give *h(x)* the feature *parented* with a word as its value. The *parented* feature ensures that no two idom links are conflated. If c is second order, then the h of

[34] For translating grammar entries this will not be necessary, since there are no produced–from constraints in SCCG grammar entries.

every argument argument is dominated by the h of its argument result, and these pairs are also added to the SUG structure's nesting list, to be described below. For every category x introduced by c, $h(x)$'s *head* terminal must have its word specified. If there are no words available in the structure, add a terminal with the empty string as its word and make it immediately dominated by some nonterminal.

An SCCG grammar entry can be mapped into an equivalent SUG grammar entry by translating each category in the set as described above. If a category is mentioned more than once in the SCCG grammar entry, then the information due to each mentioning is all stated on the same node in the SUG grammar entry. These grammar entries are equivalent in the sense that an entire grammar translated in this way will be weakly equivalent to the original grammar.

The nesting list and its effects are the slight modification of SUG mentioned previously as being necessary to translate SCCG categories into SUG structures. With this modification SUG derivations are unchanged except the pairs in the nesting list must not cross in the dominance structure of the resulting tree. In other words, there must be some total ordering of the pairs in the nesting list such that for any given pair $\langle x,y \rangle$ there is no pair $\langle z,w \rangle$ after $\langle x,y \rangle$ in the ordering, such that either z or w is between x and y in the dominance structure and not equal to x or y. Since the pairs in the nesting lists produced by the mapping specified in this section coincide with the use of nonredundant dominance relationships, this restriction essentially manifests a restriction on long distance dependencies. All the dependencies which are ruled out by this restriction are also ruled out by the Path Containment Condition, proposed as a universal linguistic constraint in [Pesetsky, 1982].

Using this translation any SCCG derivation can be simulated by an equivalent SUG derivation. For each SCCG reduction there is an SUG combination which does the same equations as the SCCG reduction. If an SCCG reduction results in a single basic category, then the associated SUG combination will result in a complete description. The difference between the SCCG reductions and the SUG combinations is that the SCCG reductions both do equations and remove the categories equated. If these are the only mentionings of the category, then the SCCG reduction in effect abstracts away from the existence of that category. The SUG combination does not do such abstraction. However, the fact that the two derivations are equivalent indicates that the abstraction could have been done in the SUG derivation without any problems. In fact, such abstraction can be done in an SUG derivation exactly when it can be done in an equivalent SCCG derivation. Thus SCCG

provides a theory of how to abstract away from characteristics of an SUG structure without thereby allowing violations of the forgotten constraints. Coordination can be handled in SUG by performing these abstractions on the coordinated structures until they are the same, then using this common abstracted structure as the result of the coordination. It is hoped that this process of equating then abstracting is indicative of syntactic processing in general, and will lead to a better understanding of constraints such as memory limitations, which also require this type of abstraction in order to conserve memory. This later topic will be discussed briefly at the end of this thesis in the section on future research.

3.7.3 Capturing Coordination in SCCG

With the extended version of CCG presented in the last section and the mapping from analyses in this formalism to SUG analyses, it is now possible to show how CCG's theory of coordination can be applied to SUG. Figures 35 to 38 give examples of coordination phenomena with their CCG, SCCG, and SUG analyses. For CCG and SCCG the coordination is allowed by the coordination schema X *and* $X \longrightarrow X$. This schema allows any two derivation structure constituents to coordinate as long as they are the same category. This analysis of coordination in SCCG can be applied to SUG through the mapping given in the previous section. Each SUG structure to be considered here is equivalent to a set of SCCG categories. These categories represent the type of the structure. Two SUG structures can coordinate if each of their types can be reduced to a common category without any equations. Since no equations are done in these reductions, they simply abstract away from certain characteristics of the structures, as mentioned above. Once a common abstraction has been found for the types of the two SUG structures, this common category can be translated back into an SUG structure and be used as the result of the coordination. In this way SCCG acts as a theory of abstraction for SUG structures, and this theory of abstraction is used to determine the common characteristics of the coordinated structures[35]. In each of the examples the common type for the coordinated SUG structures is the SCCG category which is coordinated in the SCCG derivation.

Figure 35 gives a simple example of what is typically treated as nonconstituent coordination. Since CCG defines constituency in terms of derivation structure and CCG's

[35]This definition of coordination is actually a little more restrictive than simply requiring the two structures to have a common abstraction. Here the result of the coordination must have a single SCCG category as its type. This is done to make the SUG analysis more closely follow the SCCG and CCG analyses. Loosening this constraint will be discussed in the last example in this section.

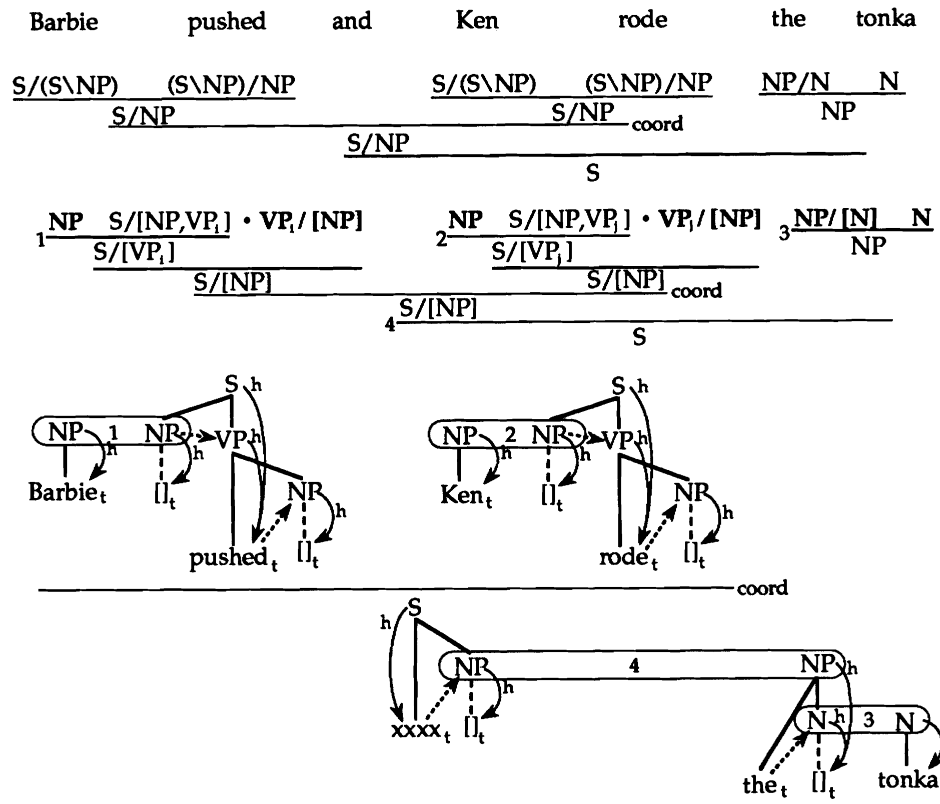


Figure 35: The CCG, SCCG, and SUG analyses of "Barbie pushed and Ken rode the tonka", in that order. In the SCCG analysis, subscripts are used to designate what categories are identical, multiple categories for the same word are specified by putting a concatenation dot between the categories, and the linear precedence constraints are the same as those shown in the SUG analysis. The reductions in the SCCG analysis which result in the equation of categories are numbered in correspondence with the equations of the associated nodes in the SUG analysis.

derivations are sufficiently flexible, this example of coordination is handled in the same way as coordination of conventional constituents. For the same reasons, SCCG can also handle this example. In the SCCG example the subject does not need a type raised entry because of the flexibility introduced with the change from directional slashes to linear precedence constraints. The only other difference is the inclusion of a VP node in the projection of the verb. Both these differences are orthogonal to the coordination analysis. The SUG analysis is a translation of the SCCG analysis into SUG structures. Each SUG grammar entry is a translation of the SCCG grammar entry in accordance with the mapping given above. The combination of the subjects' structures with those of the verbs corresponds to the SCCG reductions labeled 1 and 2, except the SUG combination only does the equations of the subject NPs, without abstracting away from their existence. Each of the resulting combined structures is then equivalent to the SCCG category for its subject plus the categories for its verb, only with the subject NPs coreferenced. These sets of categories can each be reduced without equations to the category S/[NP], analogously to that portion of the SCCG derivation. An SUG structure equivalent to this SCCG category is then used to combine with the object to produce a complete SUG structure.

Figure 36 demonstrates that the coordinated SUG structures do not have to be the same, as long as they have the same functional behavior in a derivation, as indicated by their common reduced SCCG category. The result of this coordination abstracts away from the differences between the structures and manifests some of the common characteristics of their types.

A more challenging example of nonconstituent coordination is given in figure 37. In this example a verb which subcategorizes for a sentence must be combined with the subject of that sentence without the subject's verb. In CCG this requires the subject of the subordinate clause to be type raised. The same technique can be used in SCCG, as is shown in the SCCG analysis[36]. This is translated into an SUG analysis using the ability to specify dominance relationships. The type raised NPs translate to structures with a headless S which dominates a headed NP, thus expressing the expectation for a headed S which subcategorizes for the NP. Note that reduction 3 in the SCCG analysis corresponds to two

[36] This type raised category for NP's may seem rather arbitrary, since it singles out S's, as opposed to the other things which might subcategorize for an NP. However, in a more general analysis the S category might be underspecified so as to allow any category which might subcategorize for an NP. Such a grammar entry would simply manifest the fact that all NP's are subcategorized for by something. In other words, that all NP's receive Case.

equations in the SUG analysis.

The top analysis in figure 38 shows how CCG can handle the modification of phrases internal to coordinated constituents. The categories for "the men" and "the women" are NPs which have been type raised with respect to an NP modifier. This allows these phrases to combine with their verbs before they combine with the modifying PP. Similar analyses could be given in SCCG and SUG, but in figure 38 an alternative approach is taken. The problem with the CCG analysis is that a parser would have to choose the grammar entry shown for "the men" on the basis of the PP at the end of the sentence. This would cause problems for an incremental parser. If "the men" and "the women" were simply given the category NP, then when "likes the men" and "hates the women" are each combined, these NP's are removed and thus there is nothing for "in her class" to modify. In SCCG this problem could be avoided by separating the combination rule into two parts, one which does the category equations and one which does the reduction of the two categories. This would permit the combination marked with a star in the SCCG analysis. If we then loosen the criteria for coordinatability to allow the coordination of sets of categories which are linked by common subcategories, then the example is allowed. This analysis fits nicely with SUG, since the equation half of the SCCG combination rule corresponds directly to the equation of nodes in SUG, and the reduction half corresponds to abstracting away from the existence of the node. The new definition of coordinatability in SUG would simply allow the coordination of any pair of connected tree fragments, as long as the fragments' sets of SCCG categories can each be reduced without equations to a common set of categories. This definition replaces the requirement that the coordinated phrases be reduced to a single category with the requirement that they be combined into a single tree fragment. The later notion has no correlate in CCG.

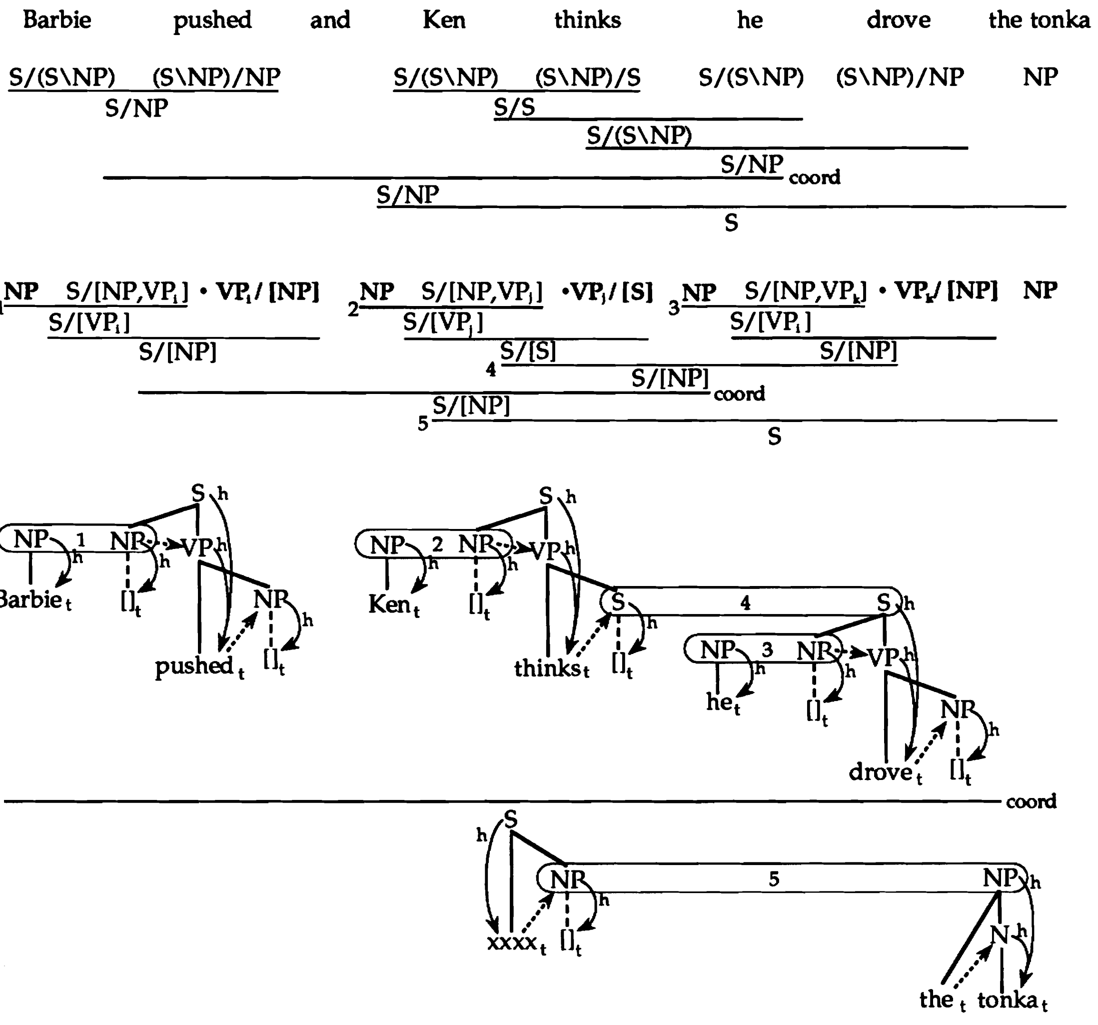


Figure 36: The CCG, SCCG, and SUG analyses of "Barbie pushed and Ken thinks he drove the tonka", in that order.

Barbie thinks Ken and knows Joe squeaks

NP (S\NP)/S S/(S\NP) (S\NP)/S S/(S\NP) S\NP

(S\NP)/(S\NP) (S\NP)/(S\NP) coord

(S\NP)/(S\NP)

S\NP

S

NP S/[NP,VP$_i$] • VP$_i$/[S] S$_j$/[S$_j$/[NP]] S/[NP,VP$_k$] • VP$_k$/[S] S$_l$/[S$_l$/[NP]] S/[NP,VP$_m$] • VP$_m$

1 S/[NP,S] 2 S/[NP,S] S/[NP]

S/[NP,S/[NP]] S/[NP,S/[NP]] coord

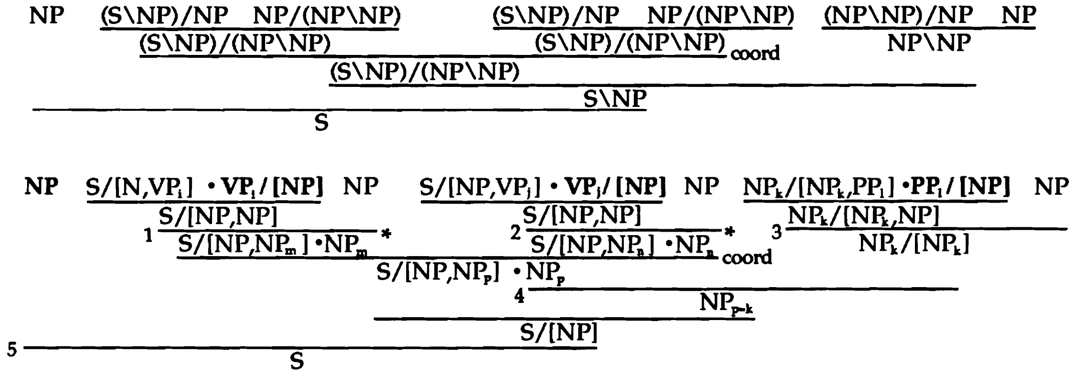
3 S/[NP,S/[NP]]

S/[NP]

4 S

Figure 37: The CCG, SCCG, and SUG analyses of "Barbie thinks Ken and knows Joe squeaks", in that order.

Barbie likes the men and hates the women in her class



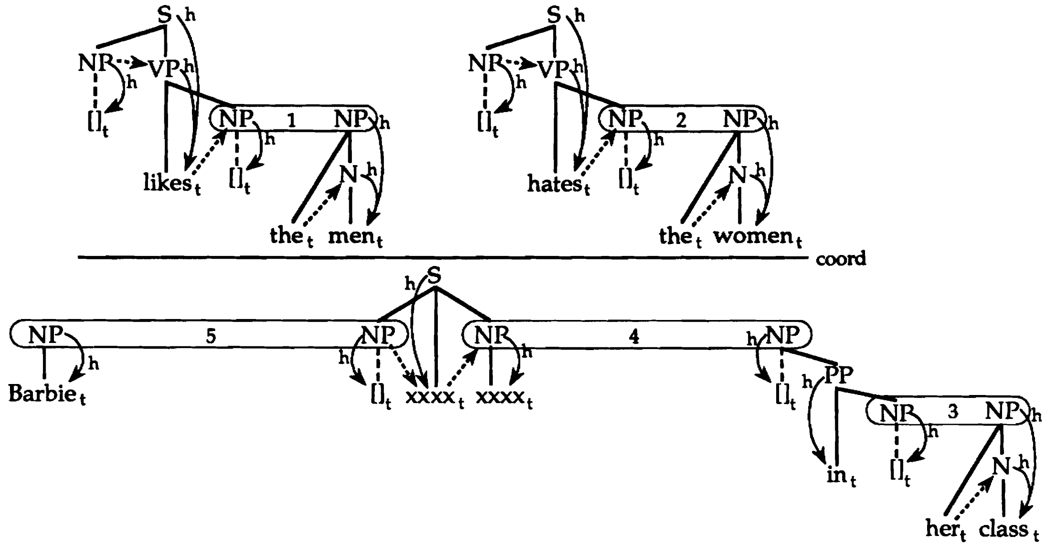


Figure 38: A CCG analysis of "Barbie likes the men and hates the women in her class" and an alternative analysis in SUG and a modified version of SCCG. The SCCG reductions marked with a star only equate the categories which would ordinarily be equated and removed. Because they are not removed the subsequent modification is possible.

Chapter 4

Conclusions and Future Directions

This thesis has presented Structure Unification Grammar and demonstrated its usefulness for representing grammatical information through a series of comparisons with other investigations into natural language. These comparisons have demonstrated that a diverse collection of insights from a diverse collection of investigations can be unified using SUG as a framework. By investigating these insights within a common formalism we can see how they interact and thus gain a better understanding of language as a whole.

The tools which allow SUG to successfully unify insights from often incompatible formalisms are SUG's perspicuous representation of phrase structure trees and SUG's ability to partially specify this information. Research into natural language has repeatedly demonstrated the usefulness of these tools, and this thesis further supports their importance. Even investigations such as CCG which purport not to need phrase structure can be interpreted in structural terms. Feature structures have been used in many formalisms to allow the partial specification of node labels. SUG differs from most of these formalisms by allowing structural relations to be equally partial. In SUG both immediate dominance and linear precedence relations can be only partially specified, and chains of immediate dominance relations can be underspecified with dominance relations.

SUG's representation of grammatical information gives it two characteristics which make the perspicuous representation of a diverse collection of constraints possible. First, SUG can partially specify constraints, thus allowing any information which is not known to be left unspecified. Second, SUG's domain of locality for specifying grammatical constraints is very large. Because dominance relationships can be specified, long distance dependencies can be specified in a single grammar entry. Also, since there are no limitations on the

sets of structural relations which can be specified, predicate–argument relationships can be specified directly in single grammar entries.

SUG's large domain of locality does a lot to permit information to be stated where it is known, but limitations on the ways structure descriptions can combine could interfere with the flexibility with which such specification can be done in a grammar. For this reason SUG allows arbitrary node equations when combining structure descriptions, provided the resulting description is satisfiable. No other information can be added, since this would undermine the ability of the grammar to constrain the possible structures. Because of this combination operation any derivation structure is possible, regardless of the phrase structure, and the set of nodes in one grammar entry may overlap arbitrarily much with those of another entry when the derivation is done. This property permits grammatical constraints to be spread across the grammar according to information dependencies, rather than according to structural configuration. This property together with the above two properties, mean that an SUG grammar can state exactly what is known, where it is known.

When an SUG derivation is done the resulting description must completely specify the information in some phrase structure tree so a unique phrase structure tree result can be found. This means all nodes except the root must have immediate parents and all the words of each terminal must be specified. These requirements can be used by the grammar writer to ensure that certain information will be specified during the course of a derivation. This provides the ability to specify obligatory arguments and ensures that all structure fragments will be used in the final description. These techniques plus the formentioned flexibility in specifying grammatical information give SUG the power and perspicuous representations necessary to unify the insights from the diverse collection of investigations discussed here.

The power of SUG is demonstrated by its ability to specify almost all the constraints specifiable in Lexical Functional Grammar. This includes the ability to constrain possible long distance dependencies, and the ability to express LFG's representation of semantic information in the feature structure labels of SUG nodes. Some of the other LFG constraints are not so easily simulated in SUG, but this seems to be an indication of the inherent complexity of enforcing these constraints.

Although the details differ, SUG's representation of grammatical information has the same basic character as that of D-Theory. They both depend heavily on the partial description of phrase structure trees. These partial descriptions allow D-Theory to do syntactic parsing incrementally and deterministically by allowing the specification of what the parser

is sure of while allowing specification of other phrase structure information to be delayed until later in the parse. SUG's use of partial specifications allow for the same degree of flexibility, thus also supporting an incremental deterministic parser.

SUG's representation also supports the type of parser proposed by Abney in [Abney, 1986]. Abney's parser is based on the linguistic notion of licensing, only extended so that all phrases must be licensed. A sentence is parsed by recovering these licensing relations. This approach combines linguistic concerns with parsing concerns because these licensing relations are both very general across languages and, when represented properly, can be easily recovered by a psychologically plausible parser. One key idea in the representation of these relations is the use of anti-relations, which are specified with the licensee rather than with the licensor. Anti-relations are used primarily for licensing adjuncts. The close relationship between licensing relations and phrase structure relations permits SUG to manifest the same information in its representation of phrase structure. Because the division of grammatical information in SUG does not have to follow any specific structural configurations, both regular licensing relations and anti-relations can be supported. Thus SUG also supports an efficient psychologically plausible parser for recovering licensing information.

The importance of some of the characteristics of SUG are demonstrated in work on Tree Adjoining Grammar. Like SUG, the data structures of a TAG grammar are phrase structure trees, and TAG has a large domain of locality for specifying grammatical constraints. TAG can state both long distance dependencies and predicate–argument relationships directly within single grammar entries, as can SUG. Linguistic work in TAG (for example [Kroch and Joshi, 1985]) has pointed out the importance of these abilities. The explicit representation of phrase structure in TAG, and SUG, is also useful because it provides for a distinction between phrase structure and derivation structure, which is important in combining the insights of CCG with those of TAG analyses and other linguistic work.

Although it is not as flexible as in SUG, Lexicalized Tree Adjoining Grammar's ability to partition constraints among grammar entries is sufficiently flexible to allow TAG grammars to be lexicalized. This allows for a more modular expression of grammatical information than in TAG. Because SUG has both LTAG's explicit representation of phrase structure and the ability to express the information dependencies expressible in LTAG, SUG can use the same analyses as LTAG in the specification of a lexicalized grammar.

Most of the investigations discussed in this thesis use a semantically based conception of phrase structure. However, the usefulness of SUG is not limited to such investigations, as

is demonstrated by its ability to incorporate insights from Combinatory Categorial Grammar. CCG proposes a notion of constituent structure which is based on coordination and extraction phenomena. Since SUG structures behave in a similar manner to the functional types of CCG, this notion of constituency can be captured in the derivation structures of SUG, while still maintaining the explicit representation of conventional phrase structure. This makes it possible to apply CCG's theory of coordination to SUG, thus allowing what is usually called nonconstituent coordination to be treated in the same way as constituent coordination.

4.1 Future Directions

The work presented in this thesis runs counter to most work on grammatical formalisms, because there is no attempt made to show that SUG constrains the possible languages. This other work, called constrained grammatical formalisms, tries to find formalisms which are powerful enough to handle natural language phenomena but not powerful enough to handle things which do not occur in natural language. This is the same objective as in linguistics, but it is usually assumed that the formalism itself will not rule out all non-natural languages, but it will give any linguistic theory specified in that formalism some of the constraints on possible languages for free. TAG and LTAG are good examples of this approach. Several constraints on long distance dependencies fall out of using TAG, when some natural assumptions are made about the form of grammar entries ([Kroch and Joshi, 1985]).

The difficulty with investigating constrained grammatical formalisms is that the constraints are implicit to the formalism and thus not easily altered if they are not desirable. Changing a constraint may involve modifying the formalism to the extent that the previous linguistic work done in that formalism needs to be significantly altered. Since the only way to test such a formalism is to try to develop a linguistic theory within that formalism, each iteration in the process of developing a constrained grammatical formalism can take a long time.

One alternative to constrained grammatical formalisms is to use a very general formalism and state the constraints explicitly on top of the formalism. This permits the linguistic work done with one set of constraints to be easily transferred to another set of constraints, since the formalism has not changed. By separating the constraints from the representation

in this way the process of investigating constraints can be significantly speeded up. This is the approach advocated here. As has been demonstrated in this thesis, SUG is a very good representation, both for specifying grammatical information and for supporting investigations into parsing. This provides a good framework for investigating computationally motivated constraints.

The work in SUG which I am currently doing falls within this approach of explicitly constraining grammatical formalisms. As an example, consider the constraint that the parser must proceed incrementally with a memory of bounded size. This constraint is motivated by the idea that the memory of the parser has similar characteristics to conscious short term memory. If the size of the description exceeds the size of the memory, then the parser must abstract away from some of the information. The CCG–like type system for SUG structures defined in section 3.7 provides a theory of how this abstraction can be done without allowing violations of the forgotten constraints. However, forgetting information will eliminate some otherwise possible parses. In particular, when a structure with a sufficiently large right frontier is built, not all the nodes on the right frontier can be remembered, so some allowable modifications and argument subcategorizations will no longer be possible. This means that in such a situation there must be some limit on how many phrases can be modified, and there must be a limit to the depth of center embedding. Natural language has both these types of constraints. In addition, a restriction on posthead modifier attachment implies the need for heavy NP shift. Given any strategy for deciding what nodes to remember for future modification, there will exist a constituent whose node will be forgotten before the last of its subcategorized arguments is parsed. If this constituent is to be modified, the modifier must come before the last argument. Thus this restriction forces the existence of heavy NP shift in order to express such modification in such contexts.

I also intend to investigate several other computationally motivated constraints. One is a more specific restriction on the memory available to the parser. With a more specific restriction on memory, specific analyses would make specific predictions about the acceptability of sentences. One candidate for this restriction is a connectionist model of short term memory proposed in [Shastri and Ajjanagadde, 1990]. It permits only a small number of entities to be remembered, but an arbitrary number of predications over those entities. Another area in need of constraint is SUG's mechanism for expressing long distance dependencies. Resolving where to equate a node which is dominated but not immediately dominated is probably the most computationally expensive part of parsing in SUG. The

linguistic constraints on long distance dependencies greatly decrease this complexity, so it is hoped that they can be "explained" in terms of efficient parsing strategies. Only future research can determine the success of this endeavor.

Bibliography

[Abney, 1986] Steven Abney. Licensing and parsing. In *Proceedings of NELS 16*, Amherst, MA, 1986.

[Brunson, 1988] Barbara A. Brunson. *A Processing Model for Warlpiri Syntax and Implications for Linguistic Theory.* Technical Report CSRI-208, University of Toronto, Toronto, Canada, 1988.

[Johnson, 1990] Mark Johnson. Expressing disjunctive and negative feature constraints with classical first–order logic. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, Pittsburgh, PA, 1990.

[Joshi, 1987a] Aravind K. Joshi. An introduction to tree adjoining grammars. In Alexis Manaster-Ramer, editor, *Mathematics of Language*, John Benjamins, Amsterdam, 1987.

[Joshi, 1987b] Aravind K. Joshi. Word-order variation in natural language generation. In *AAAI 87, Sixth National Conference on Artificial Intelligence*, pages 550–555, Seattle, Washington, July 1987.

[Joshi *et al.*, forthcoming, 1990] Aravind K. Joshi, K. Vijay-Shanker, and David Weir. The convergence of mildly context-sensitive grammatical formalisms. In Peter Sells, Stuart Shieber, and Tom Wasow, editors, *Foundational Issues in Natual Language Processing*, MIT Press, Cambridge MA, forthcoming, 1990.

[Kaplan and Bresnan, 1982] Ronald Kaplan and Joan Bresnan. Lexical functional grammar: a formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, MIT Press, 1982.

[Kroch, 1989] Anthony Kroch. Assymetries in long distance extraction in a tree adjoining grammar. In Mark Baltin and Anthony Kroch, editors, *Alternative Conceptions of Phrase Structure*, University of Chicago Press, 1989.

[Kroch and Joshi, 1985] Anthony Kroch and Aravind Joshi. *The Linguistic Relevance of Tree Adjoining Grammar.* Technical Report MS-CS-85-16, University of Pennsylvania Department of Computer and Information Sciences, 1985. To appear in Linguistics and Philosophy.

[Lambek, 1961] Joachim Lambek. On the calculus of syntactic types. In *Structure of Languageand its Mathematical Aspects. Proceedings of the Symposia in Applied Mathematics, XII*, American Mathematical Society, Providence, RI, 1961.

[Marcus, 1980] Mitchell Marcus. *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, MA, 1980.

[Marcus *et al.*, 1983] Mitchell Marcus, Donald Hindle, and Margaret Fleck. D-theory: talking about talking about trees. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Cambridge, MA, 1983.

[Pesetsky, 1982] D. Pesetsky. *Paths and Categories.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1982.

[Rounds and Kasper, 1986] William Rounds and Robert Kasper. A complete logical calculus for record structures representing linguistic information. In *IEEE Symposium on Logic and Computer Science*, 1986.

[Rounds and Manaster–Ramer, 1987] William Rounds and Alexis Manaster–Ramer. A logical version of functional grammar. In *Proceedings of the 25th Annual Meeting of the Association of Computational Linguistics*, 1987.

[Rounds, 1988] William C. Rounds. Set values for unification-based grammar formalisms and logic programming. 1988. Manuscript, CSLI and Xerox PARC.

[Schabes, 1990] Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1990.

[Schabes *et al.*, 1988] Yves Schabes, Anne Abeillé, and Aravind K. Joshi. Parsing strategies with 'lexicalized' grammars: application to tree adjoining grammars. In *Proceedings*

of the 12th International Conference on Computational Linguistics (COLING'88), Budapest, Hungary, August 1988.

[Shastri and Ajjanagadde, 1990] Lokendra Shastri and Venkat Ajjanagadde. *From Simple Associations to Systematic Reasoning: A Connectionist Representation of Rules, Variables and Dynamic Bindings.* Technical Report MS-CIS-90-05, University of Pennsylvania, Philadelphia, PA, 1990.

[Shieber, 1986] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Center for the Study of Language and Information, 1986.

[Steedman, 1987] Mark Steedman. Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory*, 5, 1987.

[Vijay-Shanker, 1987] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1987.