COMPUTATIONAL COMPLEXITY AND LEXICAL FUNCTIONAL GRAMMAR

Robert C. Berwick
MIT Artificial Intelligence Laboratory, Cambridge, MA

1. INTRODUCTION

An important goal of modern linguistic theory is to characterize as narrowly as possible the class of natural languages. An adequate linguistic theory should be broad enough to cover observed variation in human languages, and yet narrow enough to account for what might be dubbed "cognitive demands" -- among these, perhaps, the demands of learnability and parsability. If cognitive demands are to carry any real theoretical weight, then presumably a language may be a (theoretically) possible human language, and yet be "inaccessible" because it is not learnable or parsable.

Formal results along these lines have already been obtained for certain kinds of Transformational Generative Grammars: for example, Peters and Ritchie [1] showed that *Aspects*-style unrestricted transformational grammars can generate any recursively enumerable set; while Rounds [2] [3] extended this work by demonstrating that modestly restricted transformational grammars (TGs) can generate languages whose recognition time is provably exponential. (In Rounds' proof, transformations are subject to a "terminal length non-decreasing" condition, as suggested by Peters and Myhill.) Thus, in the worst case TGs generate languages whose recognition is widely recognized to be computationally intractable. Whether this "worst case" complexity analysis has any real import for actual linguistic study has been the subject of some debate (for discussion, see Chomsky [4]; Berwick and Weinberg [5]). Without resolving that controversy here however, one thing can be said: to make TGs efficiently parsable one might provide additional constraints. For instance, these additional strictures could be roughly of the sort advocated in Marcus' work on parsing [6] -- constraints specifying that TG-based languages must have parsers that meet certain "locality conditions". The Marcus' constraints apparently amount to an extension of Knuth's LR(k) locality condition [7] to a (restricted) version of a two-stack deterministic push-down automaton. (The need for LR(k)-like restrictions in order to ensure efficient processability was also recognized by Rounds [2].)

Recently, a new theory of grammar has been advanced with the explicitly stated aim of meeting the dual demands of learnability and parsability -- the Lexical Functional Grammars (LFGs) of Bresnan [8]. The theory of Lexical Functional Grammars is claimed to have all the descriptive merits of transformational grammar, but none of its computational unruliness. In LFG, there are no transformations (as classically described); the work formerly ascribed to transformations such as "passive" is shouldered by information stored in lexical entries associated with lexical items. The elimination of transformational power naturally gives rise to the hope that a lexically-based system would be computationally simpler than a transformational one.

An interesting question then is to determine, as has already been done for the case of certain brands of transformational grammar, just what the "worst case" computational complexity for the recognition of LFG languages is. If the recognition time complexity for languages generated by the basic LFG theory can be as complex as that for languages generated by a modestly restricted transformational system, then presumably LFG will also have to add additional constraints, beyond those provided in its basic theory, in order to ensure efficient parsability.

The main result of this paper is to show that certain Lexical Functional Grammars can generate languages whose recognition time *is* very likely computationally intractable, at least according to our current understanding of what is or is not rapidly solvable. Briefly, the demonstration proceeds by showing how a problem that is widely conjectured to be computationally difficult -- namely, whether there exists an assignment of 1's and 0's (or "T"s and "F"s) to the literals of a Boolean formula in conjunctive normal form that makes the formula evaluate to "1" (or "true") -- can be re-expressed as the problem of recognizing whether a particular string is or is not a member of the language generated by a certain lexical functional grammar. This "reduction" shows that in the worst case the recognition of LFG languages can be just as hard as the original Boolean satisfiability problem. Since it is widely conjectured that there cannot be a polynomial-time algorithm for satisfiability (the problem is *NP-complete*), there cannot be a polynomial-time recognition algorithm for LFG's in general either. Note that this result sharpens that in Kaplan and Bresnan [8]: there it is shown only that LFG's (weakly) generate some subset of the class of context-sensitive languages (including some strictly context-sensitive languages) and therefore, in the worst case, exponential time is known to be sufficient (though not necessary) to recognize any LFG language. The result in [8] thus does not address the question of how much time, in the worst case, is *necessary* to recognize LFG languages. The result of this paper indicates that in the worst case more than polynomial time will *probably* be necessary. (The reason for the hedge "probably" will become apparent below; it hinges upon the central unsolved conjecture of current complexity theory.) In short then, this result places the LFG languages more precisely in the complexity hierarchy.

It also turns out to be instructive to inquire into just why a lexically-based approach can turn out to be computationally difficult, and how computational tractability may be guaranteed. Advocates of lexically-based theories may have thought (and some have explicitly stated) that the banishment of transformations is a computationally wise move because transformations are computationally "expensive." Eliminate the transformations, so this casual argument goes, and one has eliminated all computational problems. Intriguingly though, when one examines the proof to be given below, the computational work done by transformations in older theories *re-emerges* in the lexical grammar as the problem of choosing between alternative categorizations for lexical items -- deciding, in a manner of speaking, whether a particular terminal item is a Noun or a Verb (as with the word *kiss* in English). This power of choice, coupled with an ability to express co-occurrence constraints over arbitrary distances across terminal tokens in a string (as in Subject-Verb number agreement) seems to be all that is required to make the recognition of LFG languages intractable. The work done by transformations has been exchanged for work done by lexical schemas, but the overall computational burden remains roughly the same.

This leaves the question posed in the opening paragraph: just what sorts of constraints on natural languages *are* required in order to ensure efficient parsability? An informal argument can be made that Marcus' work [6] provides a good first attack on just this kind of characterization. Marcus' claim was that languages easily parsed (not "garden-pathed") by people could be precisely modeled by the languages easily parsed by a certain type of restricted, deterministic, two-stack parsing machine. But this machine can be shown to be a (weak) non-canonical extension of the LR(k) grammars, as proposed by Knuth [5].

Finally, this paper will discuss the relevance of this technical result for more down-to-earth computational linguistics. As it turns out, even though general LFG's may well be computationally intractable, it is easy to imagine a variety of additional constraints for LFG theory that provide a way to sidestep around the reduction argument. All of these additional restrictions amount to making the LFG theory more restricted, in such a way that the reduction argument cannot be made to work. For example, one effective restriction is to stipulate that there can only be a finite stock of features with which to label lexical items. In any case, the moral of the story is an unsurprising one: specificity and constraints can absolve a theory of computational intractability. What may be more surprising is that the requisite locality constraints seem to be useful for a variety of theories of grammar, from transformational grammar to lexical functional grammar.

2. A REVIEW OF REDUCTION ARGUMENTS

The demonstration of the computational complexity of LFGs relies upon the standard complexity-theoretic technique of reduction. Because this method may be unfamiliar to many readers, a short review is presented immediately below; this is followed by a sketch of the reduction proper.

The idea behind the reduction technique is to take a difficult problem, in this case, the problem of determining the satisfiability of Boolean formulas in conjunctive normal form (CNF), and show that the known problem can be quickly transformed into the problem whose complexity remains to be determined, in this case, the problem of deciding whether a given string is in the language generated by a given Lexical Functional Grammar. Before the reduction proper is reviewed, some definitional groundwork must be presented. A *Boolean formula in conjunctive normal form* is a conjunction of disjunctions. A formula is *satisfiable* just in case there exists some assignment of T's and F's (or 1's and 0's) to the literals of the formula X_i that forces the evaluation of the entire formula to be T; otherwise, the formula is said to be *unsatisfiable*. For example,

$$(X_2 \vee X_3 \vee X_7) \wedge (X_1 \vee \overline{X}_2 \vee X_4) \wedge (\overline{X}_3 \vee \overline{X}_1 \vee \overline{X}_7)$$

is satisfiable, since the assignment of X_2=T (hence $\overline{X}_2$=F), X_3=F (hence $\overline{X}_3$=T), X_7=F ($\overline{X}_7$=T), X_1=T ($\overline{X}_1$=F), and X_4=F makes the whole formula evalute to "T". The reduction in the proof below uses a somewhat more restricted format where every term is comprised of the disjunction of exactly three literals, so-called 3-CNF(or "3-SAT"). This restriction entails no loss of generality (see Hopcroft and Ullman, [9], Chapter 12), since this restricted format is also NP-complete.

How does a reduction show that the LFG recognition problem must be at least as hard (computationally speaking) as the original problem of Boolean satisfiability? The answer is that any decision procedure for LFG recognition could be used as a correspondingly fast procedure for 3-CNF, as follows:
(1) Given an instance of a 3-CNF problem (the question of whether *there exists* a satisfying assignment for a given formula in 3-CNF), apply the transformational algorithm provided by the reduction; this algorithm is itself assumed to execute quickly, in polynomial time or less. The algorithm outputs a corresponding LFG decision problem, namely: (i) a lexical functional grammar and (ii) a string to be tested for membership in the language generated by the LFG. The LFG recognition problem represents or mimics the decision problem for 3-CNF in the sense that the "yes" and "no" answers to both satisfiability problem and membership problem must coincide (if there is a satisfying assignment, then the corresponding LFG decision problem should give a "yes" answer, etc.).
(2) Solve the LFG decision problem -- the string-LFG pair -- output by Step 1: if the string is in the LFG language, the original formula was satisfiable; if not, unsatisfiable.

(Note that the grammar and string so constructed depend upon just what formula is under analysis; that is, for each different CNF formula, the procedure presented above outputs a *different* LFG grammar and string combination. In the LFG case it is important to remember that "grammar" really means "grammar plus lexicon" -- as one might expect in a lexically-based theory. S. Peters has observed that a slightly different reduction allows one to keep most of the grammar fixed across all possible input formulas, constructing only different-sized lexicons for each different CNF formula; for details, see below.)

To see how a reduction can tell us something about the "worst case" time or space complexity required to recognize whether a string is or is not in an LFG language, suppose for example that the decision procedure for determining whether a string is in an LFG language takes polynomial time (that is, takes time n^k on a deterministic Turing machine, for some integer k, where n= the length of the input string). Then, since the composition of two polynomial algorithms can be readily shown to take only polynomial time (see [9] Chapter 12), the entire process sketched above, from input of the CNF formula to the decision about its satisfiability, will take only polynomial time.

However, CNF (or 3-CNF) has no *known* polynomial time algorithm, and indeed, it is considered *exceedingly* unlikely that one could exists. Therefore, it is just as unlikely that LFG recognition could be done (in general) in polynomial time.

The theory of computational complexity has a much more compact term for problems like CNF: CNF is NP-complete. This label is easily deciphered:
(1) CNF is in the class NP, that is, the class of languages that can be recognized by a non-deterministic Turing machine in polynomial time. (Hence the abbreviation "NP", for "non-deterministic polynomial". To see that CNF *is* in the class NP, note that one can simply *guess* all possible combinations of truth assignments to literals, and check each guess in polynomial time.)
(2) CNF is complete, that is, all *other* languages in the class NP can be quickly reduced to some CNF formula. (Roughly, one shows that Boolean formulas can be used to "simulate" any valid computation of a non-deterministic Turing machine.)

Since the class of problems solvable in polynomial time on a deterministic Turing machine (conventionally notated, P) is trivially contained in the class so solved by a nondeterministic Turing machine, the class P must be a subset of the class NP. A well-known, well-studied, and still open question is whther the class P is a proper subset of the class NP, that is, whether there are problems solvable in non-deterministic polynomial time that cannot be solved in deterministic polynomial time. Because all of the several thousand NP-complete problems now catalogued have so far proved recalcitrant to deterministic polynomial time solution, it is widely held that P must indeed be a proper subset of NP, and therefore that the best possible algorithms for solving NP-complete problems must take more than polynomial time (in general, the algorithms now known for such problems involve exponential combinatorial search, in one fashion or another; these are essentially methods that do no better than to brutally simulate -- deterministically, of course -- a non-deterministic machine that "guesses" possible answers.)

To repeat the force of the reduction argument then, if all LFG recognition problems were solvable in polynomial time, then the ability to quickly reduce CNF formulas to LFG recognition problems implies that all NP-complete problems would be solvable in polynomial time, and that the class P=the class NP. This possibility seems extremely remote. Hence, our assumption that there is a fast (general) procedure for recognizing whether a string is or is not in the language generated by an arbitrary LFG grammar must be false. In the terminology of complexity theory, LFG recognition must be NP-hard -- "as hard as" any other NP problem, including the NP-complete problems. This means only that LFG recognition is *at least as hard* as other NP-complete problems -- it could still be more difficult (lie in some class that contains the class NP). If one could also show that the languages generated by LFGs are in the class NP, then LFGs would be shown to be NP-complete. This paper stops short of proving this last claim, but simply conjectures that LFGs *are* in the class NP.

3. A SKETCH OF THE REDUCTION

To carry out this demonstration in detail, one must explicitly describe the transformation procedure that takes as input a formula in CNF and outputs a corresponding LFG decision problem -- a string to be tested for membership in a LFG language and the LFG itself. One must also show that this can be done quickly, in a number of steps proportional to (at most) the length of the original formula to some polynomial power. Let us dispose of the last point first. The string to be tested for membership in the LFG language will simply be the original formula, sans parentheses and logical symbols; the LFG recognition problem is to find a well-formed derivation of this string with respect to the grammar to be provided. Since the actual grammar and string one has to write down to "simulate" the CNF problem turn out to be no worse than linearly larger than the original formula, an upper bound of say, time n-cubed (where n=length of the original formula) is more than sufficient to construct a corresponding LFG; thus the reduction procedure itself can be done in polynomial time, as required. This paper will therefore have nothing further to say about the time bound on the transformation procedure.

Some caveats are in order before embarking on a proof sketch of this reduction. First of all, the relevant details of the LFG theory will have to be covered on-the-fly; see [8] for more discussion. Also, the grammar that is output by the reduction procedure will not look very much like a grammar for a natural language, although the grammatical devices that will be employed will in every way be those that are an essential part of the LFG theory. (namely, feature agreement, the lexical analog of Subject or Object "control", lexical ambiguity, and a garden variety context-free grammar.) In other words, although it is most unlikely that any natural language would encode the satisfiability problem (and hence be intractable) in just the manner outlined below, on the other hand, no "exotic" LFG machinery is used in the reduction. Indeed, some of the more powerful LFG notational formalisms -- long-distance binding, existential and negative feature operators -- have not been exploited. (An earlier proof made use of an existential operator in the feature machinery of LFG, but the reduction presented here does not.)

To make good this demonstration one must set out just what the satisfiability problem is and what the decision problem for membership in an LFG language is. Recall that a formula in conjunctive normal form is satisfiable just in case every conjunctive term evaluates to true, that is, at least one literal in each term is true. The satisfiability problem is to find an assignment of T's and F's to the literals at the bottom (note that the complement of literals is also permitted) such that the root node at the top gets the value "T" (for true). How can we get a lexical functional grammar to represent this problem? What we want is for satisfying assignments to correspond to to well-formed sentences of some corresponding LFG grammar, and non-satisfying assignments to correspond to sentences that are not well-formed, according to the LFG grammar:

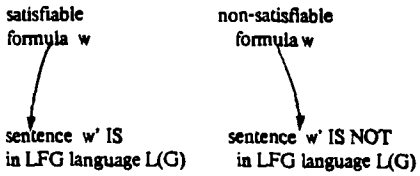


Figure 1. A Reduction Must Preserve Solutions to the Original Problem

Since one wants the satisfying/non-satisfying assignments of any particular formula to map over into well-formed/ill-formed sentences, one must obviously exploit the LFG machinery for capturing well-formedness conditions for sentences. First of all, an LFG contains a base context-free grammar. A minimal condition for a sentence (considered as a string) to be in the language generated by a lexical-functional grammar is that it can be generated by this base grammar; such a sentence is then said to have a well-formed constituent structure. For example, if the base rules included S⇒NP VP; VP⇒V NP, then (glossing over details of Noun Phrase rules) the sentence *John kissed the baby* would be well-formed but *John the baby kissed* would not. Note that this assumes, as usual, the existence of a lexicon that provides a categorization for each terminal item, e.g., that *baby* is of the category N, *kissed* is a V, etc. Importantly then, this well-formedness condition requires us to provide at least one legitimate parse tree for the candidate sentence that shows how it may be derived from the underlying LFG base context-free grammar. (There could be more than one legitimate tree if the underlying grammar is ambiguous.) Note further that the choice of categorization for a lexical item may be crucial. If *baby* was assumed to be of category V, then both sentences above would be ill-formed.

A second major component of the LFG theory is the provision for adding a set of so-called *functional equations* to the base context-free rules. These equations are used to account for that the co-occurrence restrictions that are so much a part of natural languages (e.g., Subject-Verb agreement). Roughly, one is allowed to associate features with lexical entries and with the non-terminals of specified context-free rules; these features have values. The equation machinery is used to pass features in certain ways around the parse tree, and conflicting values for the same feature are cause for rejecting a candidate analysis. To take the Subject-Verb agreement example, consider the sentence *the baby is kissing John*. The lexical entry for *baby* (considered as a Noun) might have the Number feature, with the value singular. The lexical entry for *is* might assert that the number feature of the Subject above it in the parse tree must have the value singular; meanwhile, the feature values for Subject are automatically found by another rule (associated with the Noun Phrase portion of S⇒NP VP) that grabs whatever features it finds below the NP node and copies them up above to the S node. Thus the S node gets the Subject feature, with whatever value it has passed from *baby* below -- namely, the value singular; this accords with the dicates of the verb *is*, and all is well. Similarly, in the sentence, *the boys in the band is kissing John, boys* passes up the number value plural, and this clashes with the verb's constraint; as a result this sentence is judged ill-formed:

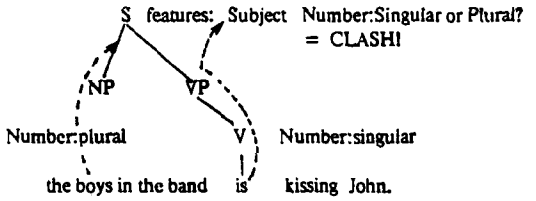


Figure 2. Co-occurrence Restrictions are Enforced by Feature Checking in an LFG.

It is important to note that the feature compatability check requires (1) a particular constituent structure tree (a parse tree); and (2) an assignment of terminal items (words) to lexical categories -- e.g., in the first Subject-Verb agreement example above, *baby* was assigned to be of the category N, a Noun. The tree is obviously required because the feature checking machinery propagates values according to the links specified by the derivation tree; the assignment of terminal items to categories is crucial because in most cases the values of features are derived from those listed in the lexical entry for an item (as the value of the number feature was derived from the lexical entry for the Noun form of *baby*). One and the same terminal item can have two distinct lexical entries, corresponding to distinct lexical categorizations; for example, *baby* can be both a Noun and a Verb. If we had picked *baby* to be a Verb, and hence had adopted whatever features are associated with the Verb entry for *baby* to be propagated up the tree, then the string that was previously well-formed, *the baby is kissing John* would now be considered deviant. If a string is ill-formed under all possible derivation trees and assignments of features from possible lexical categorizations, then that string is not in the language generated by the LFG. The possibility of multiple derivation trees and lexical categorizations (and hence multiple feature bundles) for one and the same terminal item plays a crucial rule in the reduction proof: it is intended to capture the satisfiability problem of deciding whether to give a literal X_i a value of "T" or "F".

Finally, LFG also provides a way to express the familiar patterning of grammatical relations (e.g., "Subject" and "Object") found in natural language. For example, transitive verbs must have objects. This fact of life (expressed in an *Aspects*-style transformational grammar by subcategorization restrictions) is captured in LFG by specifying a so-called PRED (for predicate) feature with a Verb; the PRED can describe what grammatical relations like "Subject" and "Object" must be filled in after feature passing has taken place in order for the analysis to be well-formed. For instance, a transitive verb like *kiss* might have the pattern, *kiss⟨(Subject)(Object)⟩*, and thus demand that the Subject and Object (now considered to be "features") have *some* value in the final analysis. The values for Subject and Object might of course be provided from some other branch of the parse tree, as provided by the feature propagation machinery; for example, the Object feature could be filled in from the Noun Phrase part of the VP expansion:

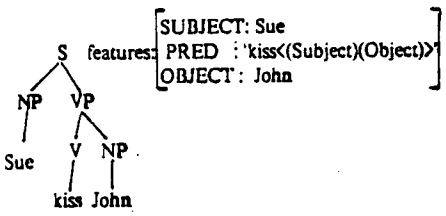


Figure 3. Predicate Templates Can Demand That a Subject or Object be Filled In.

But if the Object were not filled in, then the analysis is declared *functionally incomplete*, and is ruled out. This device is used to cast out sentences such as, *the baby kissed.*

So much for the LFG machinery that is required for the reduction proof. (There are additional capabilities in the LFG theory, such as long-distance binding, but these will not be called upon in the demonstration below.)

What then does the LFG representation of the satisfiability problem look like? Basically, there are three parts to the satisfiability problem that must be mimicked by the LFG: (1) the assignment of values to literals, e.g., X_2->"T"; X_4->"F"; (2) the co-ordination of value assignments across intervening literals in the formula; e.g., the literal X_2 can appear in several different terms, but one is not allowed to assign it the value "T" in one term and the value "F" in another (and the same goes for the complement of a literal: if X_2 has the value "T", $\overline{X}_2$ cannot have the value "T"); and (3) satisfiability must correspond to LFG well-formedness, i.e., each term has the truth value "T" just in case at least one literal in the term is assigned "T" and all terms must evaluate to "T".

Let us now go over how these components may be reproduced in an LFG, one by one.

(1) Assignments: The input string to be tested for membership in the LFG will simply be the original formula, sans parentheses and logical symbols; the terminal items are thus just a string of X_i's. Recall that the job of checking the string for well-formedness involves finding a derivation tree for the string, solving the ancillary co-occurrence equations (by feature propagation), and checking for functional completeness. Now, the context-free grammar constructed by the transformation procedure will be set up so as to generate a virtual copy of the associated formula, down to the point where literals X_i are assigned their values of "T" or "F". If the original CNF form had N terms, this part of grammar would look like:

$S \Rightarrow T_1\ T_2\ \ldots\ T_n$ (one "T" for each term)

$T_i \Rightarrow Y_i\ Y_j\ Y_k$ (one triple of Y's per term)

Several comments are in order here.

(1) The context-free base that is built depends upon the original CNF formula that is input, since the number of terms, n, varies from formula to formula. In Stanley Peters' improved version of the reduction proof, the context-free base is fixed for all formulas with the rules:

$S \Rightarrow S\ S'$

$S' \Rightarrow$ T T T or $S \Rightarrow$ T T F or T F F or T F T or...

(remaining twelve expansions that have at least one "T" in each triple)

The Peters grammar works by recursing until the right number of terms is generated (any sentences that are too long or too short cannot be matched to the input formula). Thus, the number of terms in the original CNF formula need not be explicitly encoded into the base grammar.

(2) The subscripts i,j, and k depend on the actual subscripts in the original formula.

(3) The Y_i are not terminal items, but are non-terminals.

(4) This grammar will have to be slightly modified in order for the reduction to work, as will become apparent shortly.

Note that so far there are no rules to extend the parse tree down to the level of terminal items, the X_i. The next step does this and at the same time adds the power to choose between "T" and "F" assignments to literals. One includes in the context-free base grammar two productions deriving each terminal item X_i, namely, $X_iT \Rightarrow X_i$ and $X_iF \Rightarrow X_i$, corresponding to an assignment of "T" or "F" to the formula literal X_i (it is important not to get confused here between the literals of the *formula* -- these are *terminal* elements in the lexical functional grammar -- and the literals of the *grammar* -- the non-terminal symbols.) One must also add, obviously, the rules $Y_i \Rightarrow X_iT | X_iF$, for each i, and rules corresponding to the negations of variables, $\overline{X}_iT \Rightarrow \overline{X}_i$. Note that these are not "exotic" LFG rules: exactly the same sort of rule is required in the *baby* case, i.e., $N \Rightarrow$ *baby* or $V \Rightarrow$ *baby*, corresponding to whether *baby* is a Noun or a Verb. Now, the lexical entries for the "X_iT" categorization of X_i will look very different from the "X_iF" categorization of X_i, just as one might expect the N and V forms for *baby* to be different. Here is what the entries for the two categorizations of X_i look like:

X_i: X_iT (↑truth-assignment)=T
(↑assign X_i)=T

X_i: X_iF (↑assign X_i) =F

The feature assignments for the negation of the literal X_i is simply the dual of the entries above (since the sense of "T" and "F" is reversed):

$\overline{X}_i$: $\overline{X}_iT$ (↑truth-assignment)=T
(↑assign X_i)=F

$\overline{X}_i$: $\overline{X}_iF$ (↑assign X_i) =T

The role of the additional "truth-assignment" feature will be explained below.

Figure 4. Sample Lexical Entries to Reproduce the Assignment of T's and F's to a literal X_i.

The upward-directed arrows in the entries reflect the LFG feature propagation machinery. In the case of the X_iT entry, for instance, they say to "make the Truth-assignment feature of the node above X_iT have the value "T", and make the X_i portion of the Assign feature of the node above have the value T." This feature propagation device is what reproduces the assignment of T's and F's to the CNF literals. If we have a triple of such elements, and at least one of them is expanded out to X_iT, then the feature propagation machinery of LFG will merge the common feature names into one large structure for the node above, reflecting the assignments made; moreover, the term will get a filled-in truth assignment value just in case at least one of the expansions selected an X_iT path:

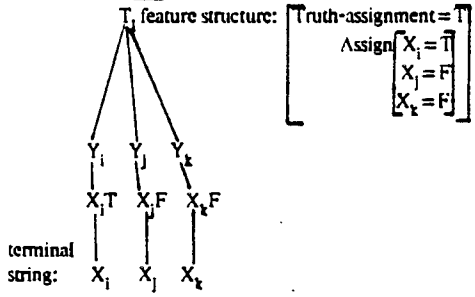


Figure 5. The LFG Feature Propagation Machinery is Used to Percolate Feature Assignments from the Lexicon.

(The features are passed transparently through the intervening Y_i nodes via the LFG "copy" device, $(\uparrow = \downarrow)$; this simply means that all the features of the node below the node to which the "copy" up-and-down arrows are attached are to be the same as those of the node above the up-and-down arrows.)

It is plain that this mechanism mimics the assignment of values to literals required by the satisfiability problem.

(2) Co-ordination of assignments: One must also guarantee that the X_i value assigned at one place in the tree is not contradicted by an X_i or X_i elsewhere. To ensure this, we use the LFG co-occurrence agreement machinery: the Assign feature-bundle is passed up from each term T_i to the highest node in the parse tree (one simply adds the $(\uparrow = \downarrow)$ notation to each T_i rule in order to indicate this). The Assign feature at this node will thus contain the union of all assign feature bundles passed up by all terms. If any X_i values conflict, then the resulting structure is judged ill-formed. Thus, only *compatible* X_i assignments are well-formed:

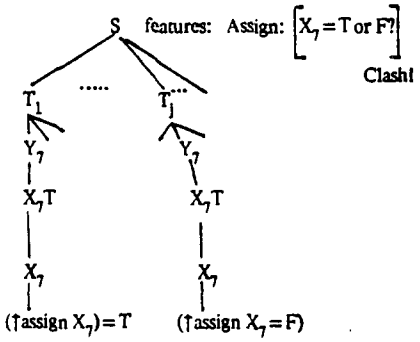


Figure 6. The Feature Compatability Machinery of LFG can Force Assignments to be Co-ordinated Across Terms.

(3) Preservation of satisfying assignments. Finally, one has to reproduce the *conjunctive* character of the 3-CNF problem -- that is, a sentence is satisfiable (well-formed) iff each term has at least one literal assigned the value "T". Part of the disjunctive character of the problem has already been encoded in the feature propagation machinery presented so far: if at least one X_i in a term T_j expands to the lexical entry X_iT, then the truth-assignment feature gets the value T. This is just as desired. If one, two, or three of the literals X_i in a term select X_iT, then T_j's truth-assignment feature is T, and the analysis is well-formed. But how do we rule out the case where all three X_i's in a term select the "F" path, X_iF? And how do we ensure that all terms have at least one T below them?

Both of these problems can be solved by resorting to the LFG functional completeness constraint. The trick will be to add a Pred feature to a "dummy" node attached to each term; the sole purpose of this feature will be to refer to the feature Truth-assignment, just as the predicate template for the transitive verb *kiss** mentions the feature Object. Since an analysis is not well-formed if the "grammatical relations" a Pred mentions are not filled in from somewhere, this will have the effect of forcing the Truth-assignment feature to get filled in *every* term. Since the "F" lexical entry does not have a Truth-assignment value, if all the X_i in a term triple select the X_iF path (all the literals are "F") then no Truth-assignment feature is ever picked up from the lexical entries, and that term never gets a Truth-assignment feature. This violates what the predicate template demands, and so the whole analysis is thrown out. (The ill-formedness is exactly analogous to the case where a transitive verb never gets an Object.) Since this condition is applied to each term, we have now guaranteed that each term must have at least one literal below it that selects the "T" path -- just as desired. To actually add the new predicate template, one simply adds a new (but dummy) branch to each term T_i, with the appropriate predicate constraint attached to it:

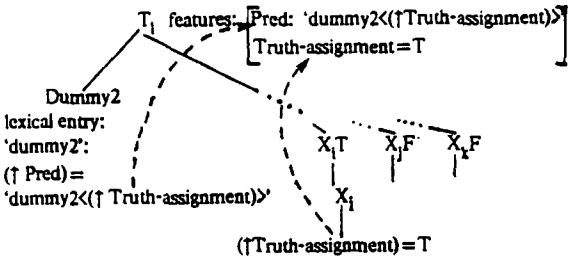


Figure 7. Predicates Can be Used to Force at least one "T" Per Term.

There is a final subtle point here: one must *prevent* the Pred and Truth-assignment features for each term from being passed up to the head "S" node. The reason is that if these features were passed up, then since the LFG machinery automatically *merges* the values of any features with the same name at the topmost node of the parse tree, the LFG machinery would force the union of the feature values for Pred and Truth-assignment over all terms in the analysis tree. The result would be that if any term had at least one "T" (hence satisfying the Truth-assignment predicate template in at least one term), then the Pred and Truth-assignment would get filled in at the topmost node as well. The string below would be well-formed if at least one term were "T", and this would amount to a disjunction of disjunctions (an "OR" of "OR"s), not quite what is sought. To eliminate this possibility, one must add a final trick: each term T_i is given *separate* Predicate, Truth-assignment, and Assign features, but *only* the Assign feature is propagated to the highest node in the parse tree as such. In contrast, the Predicate and Truth-assignment features for each term are kept "protected" from merger by storing them under *separate* feature headings labelled $T_1,...,T_n$. The means by which just the ASSIGN feature bundle is lifted out is the LFG analogue of the natural language phenomenon of Subject or Object "control", whereby *just* the features of the Subject or Object of a lower clause are lifted out of the lower clause to become the Subject or Object of a matrix sentence; the remaining features stay unmergeable because they stay protected behind the individually labelled terms.

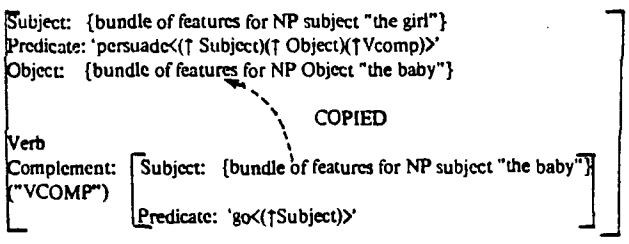
To actually "implement" this in an LFG one can add two new branches to each Term expansion in the base context-free grammar, as well as two "control" equation specifications that do the actual work of lifting the features from a lower clause to the matrix sentence:
Natural language case (from [8], pp. 43-45):

The girl persuaded the baby to go.

(part of the) lexical entry for
persuaded:

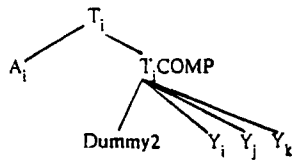
V (↑ VCOMP Subject)=(↑ Object)

The notation (↑ VCOMP Subject)=(↑ Object) -- dubbed a "control equation" -- means that the features of the Object above the V(erb) node are to be the same as those of the features of the Subject of the verb complement (VCOMP). Hence the top-most node of the parse tree eventually has a feature bundle something like:



Note how the Object features have been copied from the Subject features of the Verb Complement, via the notation described above, but the Predicate features of the Verb Complement were left behind. The satisfiability analogue of this machinery is almost identical:

Phrase structure tree:



One now attaches a "control equation" to the A_i node that forces the Assign feature bundle from the T_iCOMP side to be lifted up to get merged into the Assign feature bundle of the T_i node (and then, in turn, to become merged at the topmost node of the tree by the usual full copy up-and-down arrows):
$(\uparrow T_i\text{COMP Assign}) = (\uparrow \text{Assign})$

Note how this is just like the copying of the Subject features of a Verb Complement into the Object position of a matrix clause.

4. RELEVANCE OF COMPLEXITY RESULTS AND CONCLUSIONS

The demonstration of the previous section shows that LFGs have enough power to "simulate" a probably computationally intractable problem. But what are we to make of this result? On the positive side, a complexity result such as this one places the LFG theory more precisely in the hierarchy of complexity classes. If we conjecture, as seems reasonable, that LFG language recognition is actually in the class NP (that is, LFG recognition can be done by a non-deterministic Turing machine in polynomial time), then LFG language recognition is NP-complete. (This conjecture seems reasonable because a non-deterministic Turing machine should be able to "guess" all feature propagation solutions using its non-deterministic power -- including any "long-distance" binding solutions, an LFG device not discussed here. Since *checking* candidate solutions is quite rapid -- it can be done in n^2 time or less, as described in [8] -- recognition should be possible in polynomial time on such a machine.) Comparing this result to other known language classes, note that context-sensitive language recognition is in the class polynomial *space* ("PSPACE"), since (non-deterministic) linear bounded automata generate exactly the class of context-sensitive languages. (Non-deterministic and deterministic polynomial space classes collapse together, because of Savitch's well-known result [9] that any function computable in non-deterministic space N can be computed in deterministic space N^2.) Furthermore, the class NP is clearly a subset of PSPACE (since if a function uses Space N, it must use at least Time N), and it is suspected, but not known for certain, that NP is a proper subset of PSPACE. (This being a form of the P = NP question once again.) Our conclusion is that it is likely that LFG's generate a proper subset of the context-sensitive languages. (In [8] it is shown that this includes some strictly context-sensitive languages.) It is interesting that several other "natural" extensions of the context-free languages -- notably, the class of languages generated by the so-called "indexed grammars" -- also generate a subset of the context-sensitive languages, including those strictly context-sensitive languages shown to be generable by LFGs in [8], but are provably NP-complete (see [2] for proofs). Indeed, a cursory look at the power of the indexed grammars at least suggests that they might subsume the machinery of the LFG theory; this would be a good conjecture to check.

On the other side of the coin, how might one restrict LFG theory further so as to avoid possible intractability? Several escape hatches immediately come to mind; these will simply be listed here. Note that all of these "fixes" have the effect of adding additional constraints to further restrict the LFG theory.

1. Rule out "worst case" languages as linguistically irrelevant.
The probable computational intractability arises because co-occurrence restrictions (compatible assignment of X_i's) can be forced across arbitrary distances in the terminal string in conjunction with lexical ambiguity for each terminal item. If some device can be found in natural languages that filters out or removes such ambiguity locally (so that the choice of whether an item is "T" or "F" never depends on other items arbitrarily far away in the terminal string), or if natural languages never employ such kinds of co-occurrence restrictions, then the reduction is theoretically relevant, but linguistically irrelevant. Note that such a finding would be a positive discovery, since one would be able to further restrict the LFG theory in its attempt to characterize all and only the natural languages. This discovery would be on a par with, for example, Peters and Ritchie's observation that although the context-sensitive phrase structure rules formally advanced in linguistic theory have the power to generate non-context-free languages, that power has apparently never been used in immediate constituent analysis [11].

2. Add "locality principles" for recognition (or parsing).
One could simply stipulate that LFG languages meet some condition known to ensure efficient recognizability, e.g., Knuth's [7] LR(k) restriction, suitably extended to the case of context-sensitive languages. (See [10] for more details.)

3. Restrict the lexicon.
The reduction depends crucially upon having an infinite stock of lexical items and an infinite number of features with which to label them -- several for each literal X_i. This is necessary because as CNF formulas grow larger and larger, the number of literals can grow arbitrarily large. If, for whatever reason, the stock of lexical items or feature labels is finite, then the reduction method must fail after a certain point. This restriction seems *ad hoc* in the case of lexical items, but perhaps less so in the case of features. (Speculating, perhaps features require "grounding" in terms of other language/cognitive sub-systems -- e.g., a feature might be required to be one of a finite number of primitive "basis" elements of a hypothetical conceptual or sensori-motor cognitive system.)

ACKNOWLEDGEMENTS

I would like to thank Ron Kaplan, Ray Perrault, Christos Papadimitriou, and particularly Stanley Peters for various discussions about the contents of this paper.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-80-C-0505.

REFERENCES

[1] Peters, S. and Ritchie, R. "On the generative power of transformational grammars," Information Sciences 6, 1973, pp. 49-83.

[2] Rounds, W. "Complexity of recognition in intermediate-level languages," Proceedings of the 14th Ann. Symp. on Switching Theory and Automata, 1973.

[3] Rounds W. "A grammatical characterization of exponential-time languages," Proceedings of the 16th Ann. Symp. on Switching Theory and Automata, 1975, pp. 135-143.

[4] Chomsky, N. *Rules and Representations* New York: Columbia University Press, 1980.

[5] Berwick, R. and Weinberg, A. *The Role of Grammars in Models of Language Use*, unpublished MIT report, forthcoming, 1981.

[6] Marcus, M. A Theory of Syntactic Recognition for Natural Language, Cambridge, MA: MIT Press, 1980.

[7] Knuth, D. "On the translation of languages from left to right", Information and Control, 8, 1965, pp. 607-639.

[8] Kaplan, R. and Bresnan, J. *Lexical-functional Grammar: A Formal System for Grammatical Representation*, Cambridge, MA: MIT Cognitive Science Occasional Paper #13, 1981. (also forthcoming in Bresnan, ed., The Mental Representation of Grammatical Relations, Cambridge, MA: MIT Press, 1981.

[9] Hopcroft, J. and Ullman, J. Introduction to Automata Theory, Languages, and Computation, Reading, MA: Addison-Wesley, 1979.

[10] Berwick, R. Locality Principles and the Acquisition of Syntactic Knowledge, MIT PhD. dissertation, 1981 forthcoming.

[11] Peters, S. and Ritchie, R. *Context-sensitive immediate constituent analysis: context-free languages revisited*, Mathematical Systems Theory, 6:4, 1973, pp. 324-333.