A LOGICAL VERSION OF FUNCTIONAL GRAMMAR

William C. Rounds
University of Michigan
Xerox PARC

Alexis Manaster-Ramer
IBM T.J. Watson Research Center
Wayne State University

1 Abstract

Kay's functional-unification grammar notation [5] is a way of expressing grammars which relies on very few primitive notions. The primary syntactic structure is the *feature structure*, which can be visualised as a directed graph with arcs labeled by attributes of a constituent, and the primary structure-building operation is unification. In this paper we propose a mathematical formulation of FUG, using logic to give a precise account of the strings and the structures defined by any grammar written in this notation.

2 Introduction

Our basic approach to the problem of syntactic description is to use logical formulas to put conditions or constraints on ordering of constituents, ancestor and descendant relations, and feature attribute information in syntactic structures. The present version of our logic has predicates specifically designed for these purposes. A grammar can be considered as just a logical formula, and the structures satisfying the formula are the syntactic structures for the sentences of the language. This notion goes back to DCG's [6], but our formulation is quite different. In particular, it builds on the logic of Kasper and Rounds [3], a logic intended specifically to describe feature structures.

The formulation has several new aspects. First, it introduces the *oriented feature structure* as the primary syntactic structure. One can think of these structures as parse trees superimposed on directed graphs, although the general definition allows much more flexibility. In fact, our notation does away with the parse tree altogether.

A second aspect of the notation is its treatment of word order. Our logic allows small grammars to define free-word order languages over large vocabularies in a way not possible with standard ID/LP rules. It is not clear whether or not this treatment of word order was intended by Kay, but the issue naturally arose during the process of making this model precise. (Joshi [1] has adopted much the same conventions in tree adjunct grammar.)

A third aspect of our treatment is the use of fixed-point formulas to introduce recursion into grammars. This idea is implicit in DCG's, and has been made explicit in the logics CLFP and ILFP [9]. We give a simple way of expressing the semantics of these formulas which corresponds closely to the usual notion of grammatical derivations. There is an interesting use of *type variables* to describe syntactic categories and/or constructions.

We illustrate the power of the notation by sketching how the constructions of relational grammar [7] can be formulated in the logic. To our knowledge, this is the first attempt to interpret the relational ideas in a fully mathematical framework. Although relational networks themselves have been precisely specified, there does not seem to be a precise statement of how relational derivations take place. We do not claim that our formalization is the one intended by Postal and Perlmutter, but we do claim that our notation shows clearly the relationship of relational to transformational grammars on one hand, and to lexical-functional grammars on the other.

Finally, we prove that the satisfiability problem for our logic is undecidable. This should perhaps be an expected result, because the proof relies on simulating Turing machine computations in a grammar, and follows the standard undecidability arguments. The satisfiability problem is not quite the same problem as the *universal recognition problem*, however, and with mild conditions on derivations similar to those proposed for LFG [2], the latter problem should become decidable.

We must leave éfficiency questions unexamined in this paper. The notation has not been implemented. We view this notation as a temporary one, and anticipate that many revisions and extensions will be necessary if it is to be implemented at all. Of course, FUG itself could be considered as an implementation, but we have added the word order relations to our logic, which are not explicit in FUG.

In this paper, which is not full because of space limitations, we will give definitions and examples in Section 3; then will sketch the relational application in Section 4, and will conclude with the undecidability result and some final remarks.

3 Definitions and examples

3.1 Oriented f-structures

In this section we will describe the syntactic structures to which our logical formulas refer. The next subsection

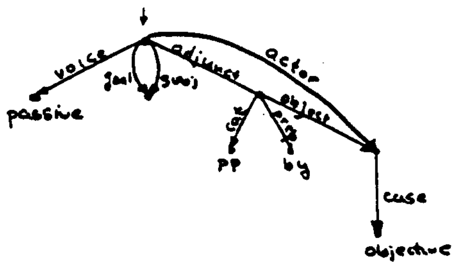


Figure 1: A typical DG.

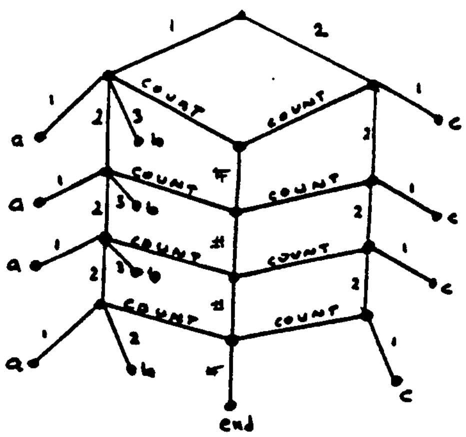


Figure 2: An oriented f-structure for $a^4b^4c^4$.

will give the logic itself. Our intent is to represent not only feature information, but also information about ordering of constituents in a single structure. We begin with the unordered version, which is the simple DG (directed graph) structure commonly used for non-disjunctive information. This is formalized as an acyclic finite automaton, in the manner of Kasper-Rounds [3]. Then we add two relations on nodes of the DG: ancestor and linear precedence. The key insight about these relations is that they are *partial*; nodes of the graph need not participate in either of the two relations. Pure feature information about a constituent need not participate in any ordering. This allows us to model the "cset" and "pattern" information of FUG, while allowing structure sharing in the usual DG representation of features.

We are basically interested in describing structures like that shown in Figure 1.

A formalism appropriate for specifying such DG structures is that of finite automata theory. A labeled DG can be regarded as a transition graph for a partially specified deterministic finite automaton. We will thus use the ordinary δ notation for the transition function of the automaton. Nodes of the graph correspond to states of the automaton, and the notation $\delta(q, x)$ implies that starting at state(node) q a transition path actually exists in the graph labeled by the sequence x, to the state $\delta(q, x)$.

Let L be a set of arc labels, and A be a set of atomic feature values. An (A, L)- automaton is a tuple

$$\mathcal{A} = (Q, \delta, q_0, \tau)$$

where Q is a finite set of states, q_0 is the initial state, L is the set of labels above, δ is a partial function from $Q \times L$ to Q, and τ is a partial function from terminating states of $\mathcal{A}$ to A. (q is terminating if $\delta(q, l)$ is undefined for all $l \in L$.) We require that $\mathcal{A}$ be connected and acyclic. The map τ specifies the atomic feature values at the final nodes of the DG. (Some of these nodes can have unspecified values, to be unified in later. This is why τ is only partial.) Let F be the set of terminating states of $\mathcal{A}$, and let $P(\mathcal{A})$ be the set of full paths of $\mathcal{A}$, namely the set $\{x \in L^* : \delta(q_0, x) \in F\}$.

Now we add the constituent ordering information to the nodes of the transition graph. Let Σ be the terminal vocabulary (the set of all possible words, morphemes, etc.) Now τ can be a partial map from Q to $\Sigma \cup A$, with the requirement that if $\tau(q) \in A$, then $q \in F$. Next, let α and $<$ be binary relations on Q, the *ancestor* and *precedence* relations. We require α to be reflexive, antisymmetric and transitive; and the relation $<$ must be irreflexive and transitive. There is no requirement that any two nodes must be related by one or the other of these relations. There *is*, however, a compatibility constraint between the two relations:

$$\forall (q, r, s, t) \in Q : (q < r) \wedge (q \; \alpha \; s) \wedge (r \; \alpha \; t) \Rightarrow s < t.$$

Note: We have required that the precedence and dominance relations be transitive. This is not a necessary requirement, and is only for elegance in stating conditions like the compatibility constraint. A better formulation of precedence for computational purposes would be the "immediate precedence" relation, which says that one constituent precedes another, with no constituents intervening. There is no obstacle to having such a relation in the logic directly.

Example. Consider the structure in Figure 2. This graph represents an oriented f-structure arising from a LFG-style grammar for the language $\{a^n b^n c^n \mid n > 1\}$.

In this example, there is an underlying CFG given by the following productions:

$$S \rightarrow TC$$

$$T \rightarrow aTb \mid ab$$

$$C \rightarrow cC \mid c.$$

The arcs labeled with numbers (1,2,3) are analogous to arcs in the derivation tree of this grammar. The root node is of "category" S, although we have not represented this information in the structure. The nodes at the ends of the arcs 1,2, and 3 are ordered left to right; in our logic this will be expressed by the formula $1 < 2 < 3$. The other arcs, labeled by COUNT and #, are feature arcs used to enforce the counting information required by the language. It is a little difficult in the graph representation to indicate the node ordering information and the ancestor information, so this will wait until the next section. Incidentally, no claim is made for the linguistic naturalness of this example!

3.2 A presentation of the logic

We will introduce the logic by continuing the example of the previous section. Consider Figure 2. Particular nodes of this structure will be referenced by the sequences of arc labels necessary to reach them from the root node. These sequences will be called *paths*. Thus the path 1 2 2 2 3 leads to an occurrence of the terminal symbol b. Then a formula of the form, say, 1 2 COUNT = 2 2 COUNT would indicate that these paths lead to the same node. This is also how we specify linear precedence: the last b precedes the first c, and this could be indicated by the formula $1\ 2\ 2\ 2\ 3 < 2\ 2\ 2\ 2\ 1$.

It should already be clear that our formulas will describe oriented f-structures. We have just illustrated two kinds of atomic formula in the logic. Compound formulas will be formed using $\wedge$ (and), and $\vee$ (or). Additionally, let l be an arc label. Then an f-structure will satisfy a formula of the form $l : \phi$, iff there is an l-transition from the root node to the root of a substructure satisfying ϕ. What we have not explained yet is how the recursive information implicit in the CFG is expressed in our logic. To do this, we introduce *type variables* as elementary formulas of the logic. In the example, these are the "category" variables S, T, and C. The grammar is given as a system of equations (more properly, equivalences), relating these variables.

We can now present a logical formula which describes the language of the previous section.

$$\begin{array}{lll}
S & \textbf{where} & \\
S & ::= & 1:T \wedge 2:C \wedge (\text{ 1 count} = \text{ 2 count}) \\
 & & \wedge\, (1 < 2) \wedge \phi_{12} \\
C & ::= & (1:c \wedge 2:C \wedge (\text{ count } \# = \text{ 2 count}) \wedge \phi_{12}) \\
 & \vee & (1:C \wedge (\text{count } \# = \text{end}) \wedge \phi_1) \\
T & ::= & (1:a \wedge 2:T \wedge 3:b \wedge (\text{ count } \# = \text{ 2 count}) \\
 & & \wedge\, (1<2) \wedge (2<3) \wedge \phi_{123}) \\
 & \vee & (1:a \wedge 2:b \\
 & & \wedge\, (\text{count } \# = \text{end}) \wedge (1<2) \wedge \phi_{12}),
\end{array}$$

where ϕ_{12} is the formula $(\epsilon\ \alpha\ 1) \wedge (\epsilon\ \alpha\ 2)$, in which ϵ is the path of length 0 referring to the initial node of the f-structure, and where the other ϕ formulas are similarly defined. (The ϕ formulas give the required dominance information.)

In this example, the set $L = \{1, 2, 3, \#, \text{count}\}$, the set $\Sigma = \{a, b, c\}$, and the set $A = \{\text{end}\}$. Thus the atomic symbol "end" does not appear as part of any derived string. It is easy to see how the structure in Figure 2 satisfies this formula. The whole structure must satisfy the formula S, which is given recursively. Thus the substructure at the end of the 1 arc from the root must satisfy the clause for T, and so forth.

It should now be clearer why we consider our logic a logic for functional grammar. Consider the FUG description in Figure 3.

According to [5, page 149], this description specifies sentences, verbs, or noun phrases. Let us call such structures "entities", and give a partial translation of this description into our logic. Create the type variables ENT, S, $VERB$, and NP. Consider the recursive formula

$$\begin{array}{lll}
ENT & \textbf{where} & \\
ENT & ::= & S \vee NP \vee VERB \\
S & ::= & \mathit{subj} : NP \wedge \mathit{pred} : VERB \\
 & & \wedge(\mathit{subj} < \mathit{pred}) \\
 & & \wedge((\mathit{scomp} : \mathit{none}) \vee (\mathit{scomp} : S \\
 & & \wedge(\mathit{pred} < \mathit{scomp})))
\end{array}$$

Notice that the category names can be represented as type variables, and that the categories NP and $VERB$ are free type variables. Given an assignment of a set of f-structures to these type variables, the type ENT will become well-specified.

A few other points need to be made concerning this example. First, our formula does not have any ancestor information in it, so the dominance relations implicit in Kay's patterns are not represented. Second, our word order conventions are not the same as Kay's. For example, in the pattern (*subj pred* $\cdots$), it is required that the subject be the very first constituent in the sentence, and that nothing intervene between the subject and predicate. To model this we would need to add the "immediately left of" predicate, because our $<$ predicate is transitive, and does not require this property. Next, Kay uses "CAT" arcs to represent category information, and considers "NP" to be an atomic value. It would be possible to do this in our logic as well, and this would perhaps not allow NPs to be unified with VERBs. However, the type variables would still be needed, because they are essential for specifying recursion. Finally, FUG has other devices for special purposes. One is the use of *nonlocal paths*, which are used at inner levels of description to refer to features of the "root node" of a DG. Our logic will not treat these, because in combination with recursion, the description of the semantics is quite complicated. The full version of the paper will have the complete semantics.

$$\left\{\begin{array}{l}\left[\begin{array}{l} cat = S \\ pattern = (subj\ pred \cdots) \\ subj = \left[\ cat = NP\ \right] \\ pred = \left[\ cat = VERB\ \right] \\ \left\{\begin{array}{l} \left[\ scomp = none\ \right] \\ \left[\begin{array}{l} pattern = (\cdots scomp) \\ scomp = \left[\ cat = S\ \right]\end{array}\right]\end{array}\right\}\end{array}\right] \\ \left[\ cat = NP\ \right] \\ \left[\ cat = VERB\ \right]\end{array}\right\}$$

Figure 3: Disjunctive specification in FUG.

3.3 The formalism

3.3.1 Syntax

We summarize the formal syntax of our logic. We postulate a set A of atomic feature names, a set L of attribute labels, and a set Σ of terminal symbols (word entries in a lexicon.) The type variables come from a set $TVAR = \{X_0, X_1, \ldots\}$. The following list gives the syntactical constructions. All but the last four items are atomic formulas.

1. NIL
2. TOP
3. X, in which $X \in TVAR$
4. a, in which $a \in A$
5. σ, in which $\sigma \in \Sigma$
6. $x < y$, in which x and $y \in L^*$
7. $x \; \alpha \; y$, in which x and $y \in L^*$
8. $[x_1, \ldots, x_n]$, in which each $x_i \in L^*$
9. $l : \phi$
10. $\phi \wedge \psi$
11. $\phi \vee \psi$
12. ψ **where** $[X_1 ::= \phi_1; \ldots X_n ::= \phi_n]$

Items (1) and (2) are the identically true and false formulas, respectively. Item (8) is the way we officially represent path equations. We could as well have used equations like $x = y$, where x and $y \in L^*$, but our definition lets us assert the simultaneous equality of a finite number of paths without writing out all the pairwise path equations. Finally, the last item (12) is the way to express recursion. It will be explained in the next subsection. Notice, however, that the keyword **where** is part of the syntax.

3.3.2 Semantics

The semantics is given with a standard Tarski definition based on the inductive structure of wffs. Formulae are satisfied by pairs $(\mathcal{A}, \rho)$, where $\mathcal{A}$ is an oriented f-structure and ρ is a mapping from type variables to sets of f-structures, called an *environment*. This is needed because free type variables can occur in formulas. Here are the official clauses in the semantics:

1. $(\mathcal{A}, \rho) \models NIL$ always;
2. $(\mathcal{A}, \rho) \models TOP$ never;
3. $(\mathcal{A}, \rho) \models X$ iff $\mathcal{A} \in \rho(X)$;
4. $(\mathcal{A}, \rho) \models a$ iff $\tau(q_0) = a$, where q_0 is the initial state of $\mathcal{A}$;
5. $(\mathcal{A}, \rho) \models \sigma$, where $\sigma \in \Sigma$, iff $\tau(q_0) = \sigma$;
6. $(\mathcal{A}, \rho) \models v < w$ iff $\delta(q_0, v) < \delta(q_0, w)$;
7. $(\mathcal{A}, \rho) \models v \; \alpha \; w$ iff $\delta(q_0, v) \; \alpha \; \delta(q_0, w)$;
8. $(\mathcal{A}, \rho) \models [x_1, \ldots, x_n]$ iff $\forall i, j : \delta(q_0, x_i) = \delta(q_0, x_j)$;
9. $(\mathcal{A}, \rho) \models l : \phi$ iff $(\mathcal{A}/l, \rho) \models \phi$, where $\mathcal{A}/l$ is the automaton $\mathcal{A}$ started at $\delta(q_0, l)$;
10. $(\mathcal{A}, \rho) \models \phi \wedge \psi$ iff $(\mathcal{A}, \rho) \models \phi$ and $(\mathcal{A}, \rho) \models \psi$;
11. $(\mathcal{A}, \rho) \models \phi \vee \psi$ similarly;
12. $(\mathcal{A}, \rho) \models \psi$ **where** $[X_1 ::= \phi_1; \ldots X_n ::= \phi_n]$ iff for some k, $(\mathcal{A}, \rho^{(k)}) \models \psi$, where $\rho^{(k)}$ is defined inductively as follows:
    - $\rho^{(0)}(X_i) = \emptyset$;
    - $\rho^{(k+1)}(X_i) = \{\mathcal{B} \mid (\mathcal{B}, \rho^{(k)}) \models \phi_i\}$,

    and where $\rho^{(k)}(X) = \rho(X)$ if $X \neq X_i$ for any i.

We need to explain the semantics of recursion. Our semantics has two presentations. The above definition is shorter to state, but it is not as intuitive as a syntactic, operational definition. In fact, our notation

$$\psi \textbf{ where } [X_1 ::= \phi_1, \ldots, X_n ::= \phi_n]$$

is meant to suggest that the Xs can be replaced by the ϕs in ψ. Of course, the ϕs may contain free occurrences of certain X variables, so we need to do this same replacement process in the system of ϕs beforehand. It turns out that the replacement process is the same as the process of carrying out grammatical derivations, but making replacements of nonterminal symbols all at once.

With this idea in mind, we can turn to the definition of replacement. Here is another advantage of our logic – replacement is nothing more than substitution of formulas for type variables. Thus, if a formula θ has distinct free type variables in the set $D = \{X_1, \ldots, X_n\}$, and $\phi_1, \ldots, \phi_n$ are formulas, then the notation

$$\theta[X_j \leftarrow \phi_j : X_j \in D]$$

denotes the simultaneous replacement of any free occurrences of the X_j in θ with the formula ϕ_j, taking care to avoid variable clashes in the usual way (ordinarily this will not be a problem.)

Now consider the formula

$$\psi \textbf{ where } [X_1 ::= \phi_1; \ldots X_n ::= \phi_n].$$

The semantics of this can be explained as follows. Let $D = \{X_1, \ldots, X_n\}$, and for each $k \geq 0$ define a set of formulas $\{\phi_i^{(k)} \mid 1 \leq i \leq n\}$. This is done inductively on k:

$$\phi_i^{(0)} = \phi_i[X \leftarrow TOP : X \in D];$$

$$\phi_i^{(k+1)} = \phi_i[X \leftarrow \phi_i^{(k)} : X \in D].$$

These formulas, which can be calculated iteratively, correspond to the derivation process.

Next, we consider the formula ψ. In most grammars, ψ will just be a "distinguished" type variable, say S. If $(\mathcal{A}, \rho)$ is a pair consisting of an automaton and an environment, then we define

$$(\mathcal{A}, \rho) \models \psi \textbf{ where } [X_1 ::= \phi_1; \ldots X_n ::= \phi_n]$$

iff for some k,

$$(\mathcal{A}, \rho) \models \psi[X_i \leftarrow \phi_i^{(k)} : X_i \in D].$$

Example. Consider the formula (derived from a regular grammar)

$$\begin{array}{lll} S & \textbf{where} & \\ S & ::= & (1 : a \wedge 2 : S) \vee (1 : b \wedge 2 : T) \vee c \\ T & ::= & (1 : b \wedge 2 : S) \vee (1 : a \wedge 2 : T) \vee d. \end{array}$$

Then, using the above substitutions, and simplifying according to the laws of Kasper-Rounds, we have

$$\begin{array}{lll} \phi_S^{(0)} & = & c\,; \\ \phi_T^{(0)} & = & d\,; \\ \phi_S^{(1)} & = & (1 : a \wedge 2 : c) \vee (1 : b \wedge 2 : d) \vee c\,; \\ \phi_T^{(1)} & = & (1 : b \wedge 2 : c) \vee (1 : a \wedge 2 : d) \vee d\,; \\ \phi_S^{(2)} & = & 1 : a \wedge 2 : (1 : a \wedge 2 : c) \vee (1 : b \wedge 2 : d) \vee c) \\ & \vee & 1 : b \wedge 2 : ((1 : b \wedge 2 : c) \vee (1 : a \wedge 2 : d) \vee d) \\ & \vee c. & \end{array}$$

The f-structures defined by the successive formulas for S correspond in a natural way to the derivation trees of the grammar underlying the example.

Next, we need to relate the official semantics to the derivational semantics just explained. This is done with the help of the following lemmas.

Lemma 1 $(\mathcal{A}, \rho) \models \phi_i^{(k)} \leftrightarrow (\mathcal{A}, \rho^{(k)}) \models \phi_i$.

Lemma 2 $(\mathcal{A}, \rho) \models \theta[X_j \leftarrow \phi_j : X_j \in D]$ *iff* $(\mathcal{A}, \rho^*) \models \theta$, *where* $\rho^*(X_i) = \{\mathcal{B} \mid (\mathcal{B}, \rho) \models \phi_i\}$, *if* $X_i \in D$, *and otherwise is* $\rho(X)$.

The proofs are omitted.

Finally, we must explain the notion of the *language* defined by ϕ, where ϕ is a logical formula. Suppose for simplicity that ϕ has no free type variables. Then the notion $\mathcal{A} \models \phi$ makes sense, and we say that a string $w \in L(\phi)$ iff for some ***subsumption-minimal*** f-structure $\mathcal{A}$, $\mathcal{A} \models \phi$, and w is compatible with $\mathcal{A}$. The notion of subsumption is explained in [8]. Briefly, we have the following definition.

Let $\mathcal{A}$ and $\mathcal{B}$ be two automata. We say $\mathcal{A} \sqsubseteq \mathcal{B}$ ($\mathcal{A}$ subsumes $\mathcal{B}$; $\mathcal{B}$ extends $\mathcal{A}$) iff there is a *homomorphism* from $\mathcal{A}$ to $\mathcal{B}$; that is, a map $h : Q_{\mathcal{A}} \rightarrow Q_{\mathcal{B}}$ such that (for all existing transitions)

1. $h(\delta_{\mathcal{A}}(q, l)) = \delta_{\mathcal{B}}(h(q), l)$;
2. $\tau(h(q)) = \tau(q)$ for all q such that $\tau(q) \in A$;
3. $h(q_{0_{\mathcal{A}}}) = q_{0_{\mathcal{B}}}$.

It can be shown that subsumption is a partial order on isomorphism classes of automata (without orderings), and that for any formula ϕ without recursion or ordering, that there are a finite number of subsumption-minimal automata satisfying it. We consider as candidate structures for the language defined by a formula, only automata which are minimal in this sense. The reason we do this is to exclude f-structures which contain terminal symbols not mentioned in a formula. For example, the formula NIL is satisfied by any f-structure, but only the minimal one, the one-node automaton, should be the principal structure defined by this formula.

By compatibility we mean the following. In an f-structure $\mathcal{A}$, restrict the ordering $<$ to the terminal symbols of $\mathcal{A}$. This ordering need not be total; it may in fact be empty. If there is an extension of this partial order on the terminal nodes to a total order such that the labeling symbols agree with the symbols labeling the positions of w, then w is compatible with $\mathcal{A}$.

This is our new way of dealing with free word order. Suppose that no precedence relations are specified in a formula. Then, minimal satisfying f-structures will have an empty $<$-relation. This implies that any permutation of the terminal symbols in such a structure will be allowed. Many other ways of defining word order can also be expressed in this logic, which enjoys an advantage over ID/LP rules in this respect.

4 Modeling Relational Grammar

Consider the relational analyses in Figures 4 and 5. These analyses, taken from [7], have much in common with functional analyses and also with transsformational ones. The present pair of networks illustrates a kind of raising construction common in the relational literature. In Figure 4, there are arc labels P, 1, and 2, representing "predicate", "subject", and "object" relations. The "c1" indicates that this analysis is at the first linguistic *stratum*, roughly like a transformational cycle. In Figure 5, we learn that at the second stratum, the predicate ("believed") is the same as at stratum 1, as is the subject. However, the object at level 2 is now "John", and the phrase "John killed the farmer" has become a "chômeur" for level 2.

The relational network is almost itself a feature structure. To make it one, we employ the trick of introducing an arc labeled with l, standing for "previous level". The conditions relating the two levels can easily be stated as path equations, as in Figure 6.

The dotted lines in Figure 6 indicate that the nodes they connect are actually identical. We can now indicate precisely other information which might be specified in a relational grammar, such as the ordering information $1 < P < 2$. This would apply to the "top level", which for Perlmutter and Postal would be the "final level", or surface level. A recursive specification would also become possible: thus

$$\begin{array}{rcl}
SENT & ::= & CLAUSE \wedge (1 < P < 2) \\
CLAUSE & ::= & 1 : NOM \wedge P : VERB \\
 & & \wedge\, 2 : (CLAUSE \vee NOM) \\
 & & \wedge\, (RAISE \vee PASSIVE \vee \ldots) \\
 & & \wedge\, l : CLAUSE \\
 & & \ldots \\
RAISE & ::= & l : 2 : CLAUSE \wedge \text{(equations in (6))} \\
 & & \ldots
\end{array}$$

This is obviously an incomplete grammar, but we think it possible to use this notation to give a complete specification of an RG and, perhaps at some stage, a computational test.

5 Undecidability

In this section we show that the problem of *satisfiability* – given a formula, decide if there is an f-structure satisfying it – is undecidable. We do this by building a formula which describes the computations of a given Turing machine. In fact, we show how to speak about the computations of an automaton with one stack (a pushdown automaton.) This is done for convenience; although the halting problem for one-stack automata is decidable, it will be clear from the construction that the computation of a two-stack machine could be simulated as well. This model is equivalent to a Turing machine – one stack represents the tape contents to the left of the TM head, and the other, the tape contents to the right. We need not simulate moves which read input, because we imagine the TM started with blank tape. The halting problem for such machines is still undecidable.

We make the following conventions about our PDA. Moves are of two kinds:

- $\mathtt{q_i}$: `push b; go to` $\mathtt{q_j}$;
- $\mathtt{q_i}$: `pop stack; if a go to` $\mathtt{q_j}$ `else go to` $\mathtt{q_k}$.

The machine has a two-character stack alphabet $\{a, b\}$. (In the push instruction, of course pushing "a" is allowed.) If the machine attempts to pop an empty stack, it cannot continue. There is one final state $\mathtt{q_f}$. The machine halts sucessfully in this and only this state. We reduce the halting problem for this machine to the satisfiability problem for our logic.

```
Atoms: "none" --- bookkeeping marker
                    for telling what
                     is in the stack
          q0, q1 ,..., qn ---   one for
                               each state

Labels: a, b    ---     for describing
                         stack contents
          s -- pointer to top of stack
          next --- value of next state
          p     --- pointer to previous
                    stack configuration

Type variables:
          CONF -- structure represents
               a machine configuration
          INIT, FINAL --configurations
                    at start and finish
          Q0,...,QN: property of being
               in one of these states
```

The simulation proceeds as in the relational grammar example. Each configuration of the stack corresponds to a level in an RG derivation. Initially, the stack is empty. Thus we put

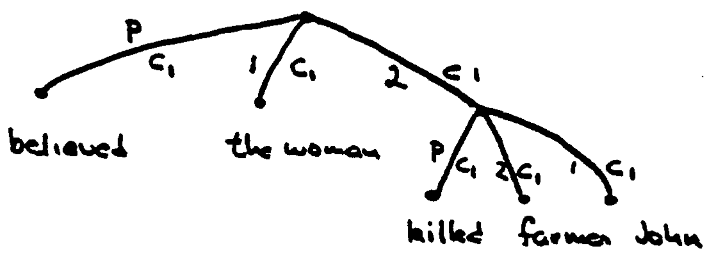


Figure 4: Network for *The woman believed that John killed the farmer.*

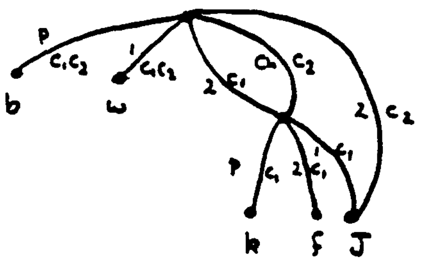
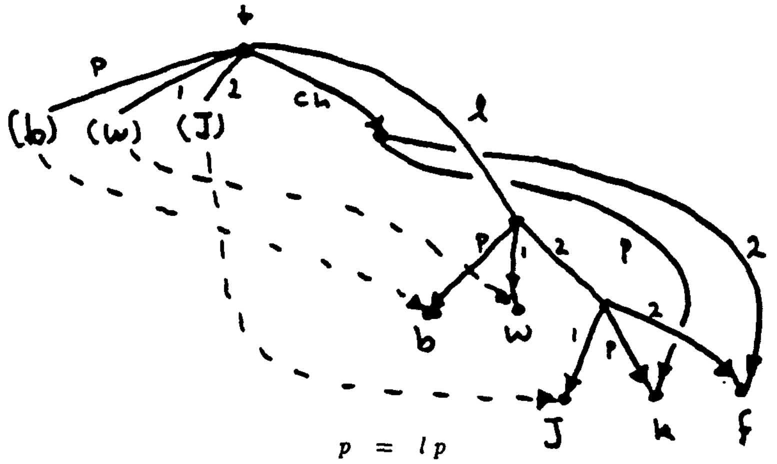


Figure 5: Network for *The woman believed John to have killed the farmer.*



$$\begin{aligned} p &= l\,p \\ 1 &= l\,1 \\ 2 &= l\,2\,1 \\ Cho\,P &= l\,2\,P \\ Cho\,2 &= l\,2\,2 \end{aligned}$$

Figure 6: Representing Figure 5 as an f-structure.

$$\mathtt{INIT} ::= \mathtt{s} : (\mathtt{b} : \mathtt{none} \wedge \mathtt{a} : \mathtt{none}) \wedge \mathtt{next} : \mathtt{q0}.$$

Then we describe standard configurations:

$$\mathtt{CONF} ::= \mathtt{INIT} \vee (\mathtt{p} : \mathtt{CONF} \wedge (\mathtt{Q0} \vee \ldots \vee \mathtt{QN})).$$

Next, we show how configurations are updated, depending on the move rules. If $\mathtt{q_i}$ is push b; go to $\mathtt{q_j}$, then we write

$$\mathtt{QI} ::= \mathtt{next} : \mathtt{qj} \wedge \mathtt{p} : \mathtt{next} : \mathtt{qi} \wedge \mathtt{s} : \mathtt{a} : \mathtt{none} \wedge \mathtt{s}\ \mathtt{b} = \mathtt{p}\ \mathtt{s}.$$

The last clause tells us that the current stack contents, after finding a "b" on top, is the same as the previous contents. The "a: none" clause guarantees that only a "b" is found on the DG representing the stack. The second clause enforces a consistent state transition from the previous configuration, and the first clause says what the next state should be.

If $\mathtt{q_i}$ is

pop stack; if a go to $\mathtt{q_j}$ else go to $\mathtt{q_k}$,

then we write the following.

$$\begin{array}{lll} \mathtt{QI} & ::= & \mathtt{p} : \mathtt{next} : \mathtt{qi} \\ & \wedge & ((\mathtt{s} = \mathtt{p}\ \mathtt{s}\ \mathtt{a} \wedge \mathtt{next} : \mathtt{qj} \wedge \mathtt{p} : \mathtt{s} : \mathtt{b} : \mathtt{none}) \\ & & \vee(\mathtt{s} = \mathtt{p}\ \mathtt{s}\ \mathtt{b} \wedge \mathtt{next} : \mathtt{qk} \wedge \mathtt{p} : \mathtt{s} : \mathtt{a} : \mathtt{none})) \end{array}$$

For the last configuration, we put

$$\mathtt{QF} ::= \mathtt{CONF} \wedge \mathtt{p} : \mathtt{next} : \mathtt{qf}.$$

We take QF as the "distinguished predicate" of our scheme.

It should be clear that this formula, which is a big where-formula, is satisfiable iff the machine reaches state $\mathtt{q_f}$.

6 Conclusion

It would be desirable to use the notation provided by our logic to state substantive principles of particular linguistic theories. Consider, for example, Kashket's parser for Warlpiri [4], which is based on GB theory. For languages like Warlpiri, we might be able to say that linear order is only explicitly represented at the morphemic level, and not at the phrase level. This would translate into a constraint on the kinds of logical formulas we could use to describe such languages: the $<$ relation could only be used as a relation between nodes of the *MORPHEME* type. Given such a condition on formulas, it might then be possible to prove complexity results which were more positive than a general undecidability theorem. Similar remarks hold for theories like relational grammar, in which many such constraints have been studied. We hope that logical tools will provide a way to classify these empirically motivated conditions.

References

[1] Joshi, A. , K. Vijay-Shanker, and D. Weir, The Convergence of Mildly Context-Sensitive Grammar Formalisms. To appear in T. Wasow and P. Sells, ed. "The Processing of Linguistic Structure", MIT Press.

[2] Kaplan, R. and J. Bresnan, LFG: a Formal System for Grammatical Representation, in Bresnan, ed. *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, 1982, 173-281.

[3] Kasper, R. and W. Rounds, A Logical Semantics for Feature Structures, *Proceedings of 24th ACL Annual Meeting*, June 1986.

[4] Kashket, M. Parsing a free word order language: Warlpiri. *Proc. 24th Ann. Meeting of ACL*, 1986, 60-66.

[5] Kay, M. Functional Grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley Linguistics Society, Berkeley, California, February 17-19, 1979.

[6] Pereira, F.C.N., and D. Warren, Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence* 13, (1980), 231-278.

[7] Perlmutter, D. M., Relational Grammar, in *Syntax and Semantics, vol. 13: Current Approaches to Syntax*, Academic Press, 1980.

[8] Rounds, W. C. and R. Kasper. A Complete Logical Calculus for Record Structures Representing Linguistic Information. *IEEE Symposium on Logic in Computer Science*, June, 1986.

[9] Rounds, W., LFP: A Formalism for Linguistic Descriptions and an Analysis of its Complexity, *Computational Linguistics*, to appear.