AUGMENTED CATEGORIAL GRAMMAR WITH ITS COMPUTATIONAL IMPLEMENTATION*

Kiyong Lee, Suson Yoo
Key-Sun Choi, Sangki Han

A grammatical system called *Augmented Categorial Grammar* (ACG) is proposed to construct a computationally tractable categorial grammar that can adequately treat some syntactic and semantic problems involving grammatical relations, semantic roles, control, and unbounded dependency in English and to use it in our subsequent work as a basic framework for developing situation semantics for natural languages. ACG is claimed to be a monostratal context-free grammar compatible with some current versions of phrase structure grammars. An implementation for a parsing system is presented to test the computational tractability of ACG. Its program is written in Prolog.

1. Introduction

The purpose of this paper is to construct a computationally tractable categorial grammar that can adequately treat some syntactic and semantic problems involving grammatical relations, semantic roles, control, and unbounded dependency in English and to use it in our subsequent work as a basic framework for developing situation semantics for natural languages. This grammar is claimed to be a monostratal context-free grammar compatible with, if not preferable to, some current versions of phrase structure grammars. At this stage, however, we are more interested in explicitly formulating some basic notions of our grammatical system called ACG than in fully testing its descriptive adequacy or providing various supporting arguments.

The present paper consists of two main sections: In section 2, we establish ACG by defining the notions of category, subcategory, matching, and generalized functional application and then, in section 3, implement the entire system in a programming language Prolog (Clocksin & Mellish 1985; Pereira & Warren 1980) to test its computational tractability.

2. Augmented Categorial Grammar

ACG, augmented categorial grammar, is a version of categorial grammar

* Supported by a grant from the Asan Foundation, Seoul. An earlier version of this paper was presented at the 1986 Annual Linguistic Conference sponsored by Language Research Institute, Seoul National University. An extensive revision of ACG has been made by the first author Kiyong Lee during his stay at CSLI, Stanford University, as Fulbright Visiting Scholar. He wishes to express his thanks to both Fulbright Commission/CIES and CSLI for their support.

based on the notion of *functor* and various other notions prevalent in unification-based grammars, especially HPSG (Pollard 1985 ; Sag & Pollard in preparation). Related to the notion of functor is the generalized notion of functional application that plays a unique role in syntactic operations. Unlike some of its predecessors in Montague grammar, ACG admits no transformational operations as such but only those computationally tractable operations that may be subsumed under the operation of generalized functional application.

ACG differs from classical Montague semantics in two respects. First, it is information-based. It attempts to go beyond a simple discussion of truth conditions but to make serious inquiries into what kind of information is conveyed by statements or questions and under what condition it is conveyed. Secondly, it attempts to interpret anaphora, scope ambiguity, and other related problems within a framework of discourse understanding. ACG, in short, adopts a situation-theoretic approach to the representation of semantic information. However, in this paper, we say very little of these aspects of ACG.

2.1. Syntactic Categories

ACG admits three types of categories: basic, functor, and storage categories.

(1) Categories
  a. Basic categories:
    A (finite) set of feature-value pairs is a category.
  b. Functor categories:
    If C is a sequence of categories C_i and B is a category, then (C,B) is a category.
  c. Storage categories:
    If C is a sequence of categories $(C_1,\cdots,C_n)$ and B is a category and, furthermore, if C_j is a j-th category in C such that its INDEX feature is defined, then $(C\{j\}, B)$ is a category,
    where $C\{j\}=(C_1,\cdots,C_{j-1},\{C_j\}, C_{j+1},\cdots,C_n)$.

Unlike GPSG or HPSG, no syntactic feature in ACG is category-valued. ACG admits only a finite number of atom-valued features. Hence, the set of basic categories is finite. From this finite list, we treat features N, V, and MAX as primary features and use them to define some basic categories each of which usually serves as argument to a functor category. Here we introduce the most commonly used four basic categories, S, NP, N, and PP, whose feature-value pairs are specified as below:

(2)
  a. $S=\{(N,-), (V,+), (MAX,+)\}$

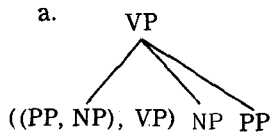
b. NP={(N,+), (V,−), (MAX,+)}
c. N={(N,+), (V,−), (MAX,−)}
d. PP={(N,−), (V,−), (MAX,+)}

Categories NP and N have the same values for the features N and V, but differ from each other with respect to the feature MAX. Although ACG, strictly speaking, does not follow any kind of X-bar theory, it distinguishes maximal phrases (S, NP, and PP) from non-maximal ones (N). On the other hand, N(*woman*) and N' (*fat woman*) are not distinguished from each other, for they are both non-maximal categories. Note that this notion of maximality is different from that of saturation discussed in Pollard (1986). Since they do not need any complements, maximal categories S and NP are saturated. On the other hand, expressions of the non-maximal category N can either be saturated or non-saturated, depending upon whether or not they need complements.

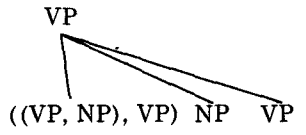
Verbal expressions belong to a functor category, V or VP. The transitive verb, say *love*, belongs to a V category defined as (NP, VP), while VP is defined as (NP,S). In Montague semantics, these categories should be interpreted as functions which take NP-denotations as argument.

We also have functor categories like ((PP,NP),VP) or ((VP,NP),VP) that are interpreted as functions each taking a pair of arguments. The verb *put*, for instance, belongs to the former, while the verb *persuade* belongs to the latter type. Unlike classical categorial grammar, our augmented CG admits unordered local trees with branches more than two.[1] Both of the above categories admit local trees with three branches:

(3) a.

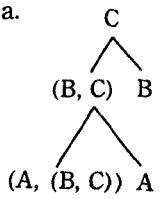


b.

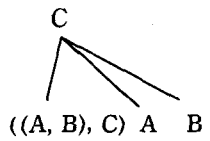


In general, both the categories (A,(B,C)) and ((A,B),C) are well-defined functor categories in ACG, if (B,C) and (A,B) are well-defined. They, however, result in two different tree structures.

(4) a.



b.



[1] Order is introduced into trees as a separate list of statements on linear precdence among sister nodes as in GPSG.

In (4a), A is not a daughter of C, but of (B,C). B is C's daughter, but A is its granddaughter. On the other hand, in (4b), both A and B are C's daughters.

As in HPSG, INDEX is introduced into ACG as a syntactic feature whose values range over parameters or natural numbers. It will be shown that the specification of INDEX makes it possible to interrelate SYNTAX with SEMANTICS. In general, a category is indexed if it has a semantic role. Since INDEX is a non-primary feature, its specification subcategorizes categories. An indexed NP[⟨INDEX,1⟩], for instance, is a subcategory of NP with an additional feature specification.[2]

Given a functor category, ACG can create a storage and put a category in it if the category is indexed with a parameter or a natural number. A stored category is enclosed by curly brackets. It may contrast with the SLASH feature in GPSG or HPSG, playing the key role in accounting for unbounded dependency in WH-questions, relativization or topicalization in English. For example, given a category (5a), we may create a storage in it as in (5b):

(5)
a. ((PP,NP), (NP, S))
b. ((PP,{NP}),(NP,S))

The verb *put* may belong to these categories. In the former case, it takes two complements PP and NP to yield (NP,S), that is, VP. In the latter case, it takes only one complement PP to form an expression of category VP with a missing NP in an object position.

2.2. Lexicon

Each entry in the lexicon contains the fullest possible amount of information concerning the syntactic, semantic, and phonological properties of a basic linguistic sign or a lexical item. This sign is represented in a matrix form divided into three components, *syntax, semantics,* and *phonology*:

(1)
SYNTAX=CATEGORY
SEMANTICS=In SITUATION
PHONOLOGY=SPELLING

The syntactic component of each matrix specifies the syntactic category which the lexical item belongs to, the semantic component gives partial description of a situation or a situation type in which the object denoted by the

[2] We shall separate the representation of primary feature specifications from that of non-primary ones by the use of square brackets: The primary specifications are represented to the left of the brackets, while the non-primary ones are enclosed by the brackets.

lexical item plays its role, and the phonological component gives information concerning how the word is pronounced. But as to the phonological component, we have little to say and so will simply give spellings.

As a very simple example, consider how a proper name like *Paul* is represented in the lexicon.

(2)

SYNTAX=CATEGORY=NP[⟨INDEX,x⟩, ⟨AGR,s3⟩, ⟨GENDER, masc⟩]
SEMANTICS=in SIT : ⟨⟨of-IND,x;1⟩⟩
⟨⟨=,x,PAUL;1⟩⟩
or simply ⟨⟨IND.x=PAUL;1⟩⟩
where IND is the type of individuals
PHONOLOGY=Paul

This matrix is interpreted as giving information about the lexical item *Paul* that it is a 3rd person singular masculine NP, denotes an individual x who is Paul, and is spelled 'Paul'.[3]

In actual practice, we try to simplify our representations. First, taking into consideration that each lexical sign consists of the syntactic, semantic, and phonological components, we can omit these labels from each matrix. We can also omit the label 'CATEGORY', since the first row in each matrix represents a syntactic category. Some syntactic features like INDEX, AGR, CASE, and GENDER need not be represented, for these features are uniquely predictable on the basis of their associated values. The label 'in SIT' can also be deleted, for each SEMANTICS is partial description of a situation or a situation type SIT. With these simplifications. the above matrix may be represented as:

(3)

NP[x,s3,masc]
⟨⟨IND.x=PAUL;1⟩⟩
Paul

Note here that the value x of INDEX interrelates SYNTAX and SEMANTICS, thus giving the information that the lexical item *Paul* is a 3rd person singular masculine NP denoting an individual PAUL.

A pronoun like *she* is associated with a matrix like:

(4) NP[x,s3,nom,fem]
⟨⟨FEMALE, IND.x;1⟩⟩

[3] We use ⟨⟨R,IND.x;1⟩⟩ as an abbreviatory description of two consistent states of affairs, ⟨⟨of-IND,x;1⟩⟩ (x is of the type INDividual) and ⟨⟨R,x;1⟩⟩.

<<=,x,IND.d(x);1>>
where IND.d(x) is some individual **b** assigned to the parameter **x** by the discourse function d.
she

This matrix shall be interpreted as giving information about the word *she* that it is a 3rd person singular nominative feminine NP denoting a female individual **x** whose reference is determined in a discourse situation d. Here we assume that there is a well-defined discourse related function that assigns individuals to parameters.

Now we introduce two indefinite and definite articles or singular determiners *a, an* and *the*:

(5)
(N,NP[x])
<<EXIST,x;1>>
an /before a vowel;
a /otherwise

(6)
(N,NP[x])
<<UNIQUE,x;1>>
the

Here the articles are treated as functors from N to NP associated with an unsaturated state of affairs in which an object x exists. Furthermore, in the case of the definite article this x should be a unique object. Note here that the semantic type of an object x is not specified. It can either be an individual, a property or anything else.

A common noun like *kitten* is a non-maximal N category denoting a property of individuals. This information is provided by the following matrix:

(7)
N[x]
<<KITTEN, IND.x;1>>
kitten

Here again the index specification x interrelates SYNTAX with SEMANTICS, giving the information that the linguistic sign *kitten* of a syntactic category N denotes the property KITTEN of a certain individual object x.

An adjective like *female* has the following representation:

(8)
(N,N[x])
⟨⟨FEMALE,IND.x;1⟩⟩
female

This lexical item belongs to the functor category (N,N[x]) and denotes the property FEMALE of an individual x.

We now treat verbs like *is, loves, got.* The copula *is* here is treated an expression of the functor category as represented below:

(9)
(NP[y,s3],(NP[x,s3], S[fin]))
⟨⟨x=y;1⟩⟩
is

Here the copular verb is interpreted as denoting an identity relation between two parameters x and y.

The transitive verb *loves* is represented as below:

(10)
(NP[2,acc], (NP[1,s3,nom], S[fin]))
⟨⟨LOVE, IND.1,IND.2;1⟩⟩
loves

This verb takes an accusative NP to form a VP and then a nominative NP to form a finite S. It is interpreted as describing a state of affairs in which two individuals have a LOVE relation, each playing a role either as a lover(1) or as a one who is loved.

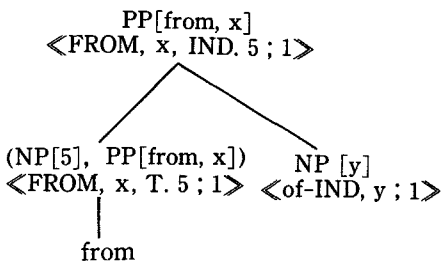
Finally, the verb *got* may have a PP complement as shown below:[4]

(11)
((PP[from,2], NP[2]), (NP[1], S[fin]))
⟨⟨GET,IND.1,IND.2;1⟩⟩
⟨⟨FROM,IND.2,x;1⟩⟩
got

Note here that PP[from,x] has the following tree structure:

[4] Here we ignored the specification of temporal location expressed by the past tense form of the verb *got.*

(12)



The preposition *from* is of the functor category taking an NP object to form a PP. It is interpreted as describing a relation of FROM between some object x and another object with index 5 whose type T is unspecified. This T is anchored to IND when the preposition combines with an NP which is interpreted as denoting an object of the IND type.

An interrogative pronoun *who* is an NP with a feature specification **que**(stion).

(13)

NP[x, nom,person,que]
≪WHO,IND.x;1≫
who

This interrogative pronoun is interpreted as describing a state of affairs in which an individual x has the property WHO. On the other hand, a relative pronoun *who* has a feature specification **rel**(ative).

(14)

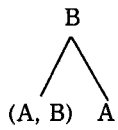
(({NP[x,nom,person,rel]}, S), (NP[x], NP[y]))
≪x=x;1≫
who

Relative pronouns are not of a basic category NP, but of a functor category with a storage. They combine with an S expression which lacks an NP to form relative clauses. Note that relative clauses are treated here as belonging to the category (NP[x], NP[y]) which we think better reflect their syntactic properties. Note also that relative pronouns are interpreted as describing a fact which always holds and thus add nothing to SEMANTICS.

2.3. Syntax: Generalized Functional Application

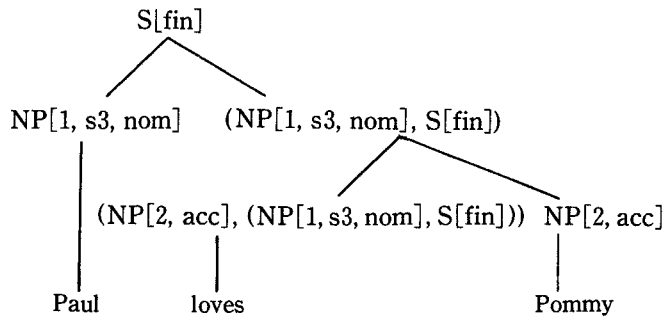
In this section we attempt to generalize the notion of functional application which is a basic syntactic operation of categorial grammar. Functional application (FA) is an operation of matching and cancellation as illustrated below:

(1)



The left element A of the functor category (A,B) *matches* with its argument category A and is thus *cancelled* to yield a category B. The following is a typical example:

(2)



In this derivation, the matching condition which we assume to be the identity of two categories, the leftmost category in a functor and its corresponding argument category, is met by anchoring the index parameters of NP *Pommy* and NP *Paul* to 2 and 1, respectively, and also assigning them case values **acc** and **nom**, respectively. The anchoring of index parameters here is forced by the basic matching frame provided by the functor category of the verb *loves*.

The operation illustrated by (2) looks straightforward, but not so simple as we expect to be as we examine its matching conditions. Suppose we try to analyze the following sentence:

(3) Who loves her?

In the lexicon, each of the words in (3) should have its category specified as below:

(4) a. who: NP[x,nom,s3,person,que]
b. loves: (NP[2,acc], (NP[1,nom], S[fin]))
c. her: NP[x,acc,s3,person,fem]

Note here that the functor category of the verb *loves* will force the index parameters x of its argument NP's *who* and *her* to be anchored to the index values 1 and 2, respectively. But then we find extra feature specifications like

person, que, or **fem** in these NP's. Their presence fails to satisfy the identity condition of matching. If we adhere to this strict version of matching, we cannot apply functional application (FA) to derive sentences like (3) on the basis of (4b).

In order to deal with this problem, ACG introduces the notion of generalized functional application by defining a set of conditions on forcing, matching and cancellation. We first define the condition of forcing:

(5) **Forcing**
Given two categories of the form (A,B) and A', where A and A' are identical except that A' is underspecified with respect to non-primary features, the functor category (A,B) forces its argument category A' to be identical with A either by anchoring parameters or adding further feature specifications.

In getting the tree (2), we have already *forced* the parameter of each argument NP to be set to the value either 1 or 2. By forcing, on the basis of (4b), we can also anchor the index parameter of (4c) as in (6):

(6) her: NP[2,acc,s3,person,fem]

But even by this type of forcing we find the category of NP *her* in (5) not exactly the same as, but more feature-specified than, the category **NP[2,acc]** in (4b). So we have a problem.

Mutual forcing might be a conceivable solution to this problem, but we do not follow this course. Instead we relax the identity condition of matching by introducing an auxiliary notion of *subcategory*.

(7) **Subcategory**
Given two categories A and A' which are identical with respect to primary feature specifications, A' is a subcategory of A if A is a subset A'.

According to this definition, the category (6), namely **NP[2,acc,s3,person,fem]** is a subcategory of the category **NP[2,acc]**. In general, subcategories are more feature-specified than their corresponding categories. But it should be noted that a subcategory relation is not a superset relation because it presupposes the identity of values in the two related categories with respect to their primary features, N, V, and MAX.

We can now define matching to be a subcategory relation.

(8) **Category Matching**
Category A matches with category A', if A' is a subcategory of A.

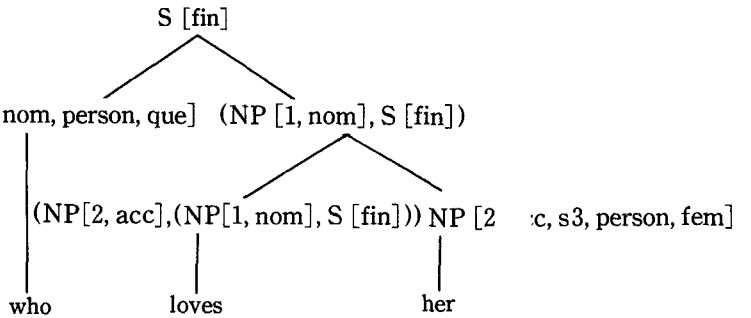
Note here that matching unlike identity is no longer a symmetric relation. We take the leftmost category A of a functor category (A,B) to match with its candidate argument category A', but not vice versa.

Using this definition, we can then generalize the notion of functional application as below:

(9) **Generalized Functional Application** (tentative)
GFA((A,B),A')=B iff A matches with A'.

This then admits a tree like the following:

(10)



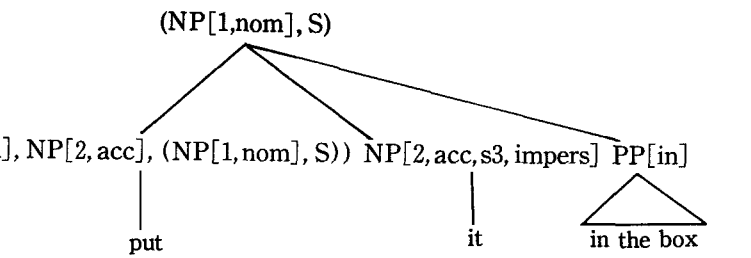
GFA has applied here twice, each time satisfying the matching condition newly defined.

But since ACG admits multi-branch trees, the matching condition as formulated in (8) need be extended to include cases where one sequence of categories may match with another sequence.

(11) **Sequential Matching**
Given two sequences of categories of the same length n,
$Z=(C_1,\cdots,C_n)$ and $Z'=(C'_1,\cdots,C'_n)$,
Z matches with Z' if every category C_i in Z matches with a unique category C'_1 in Z'.

This then admits a tree like the following:

(12)



Here the category sequence **(PP[in], NP[2,acc])** matches its argument sequence **(PP[in], NP[2,acc,s3,impers])**, for each category in the second sequence is a subcategory of its corresponding category in the first sequence. Note in particular that **NP[2,acc,s3,impers]** is a subcategory of **NP[2,acc]**. Note also that the precedence relation among complements as represented in trees, say (12), does not necessarily follow the order of their occurrences in the argument sequence Z'.

The matching condition (10), however, is still too restricted to account for cases involving storages. Suppose we try to analyze a WH-question like:

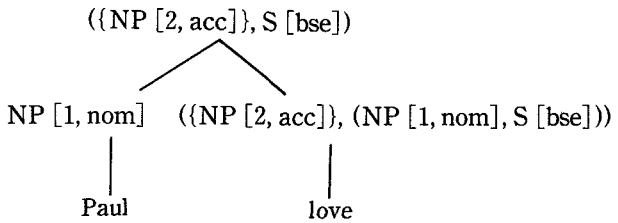
(13) Whom does Paul love?

We should have at least the following syntactic information stored in the lexicon:

(14) a. whom: NP[x,acc,person,que]
b. does: (S[bse], S[fin,que])
c. Paul: NP[x,s3,masc]
d. love : ({NP[2,acc]}, (NP[1,nom], S[bse]))

But GFA as currently formulated fails to combine *Paul* with *love* and to admit a tree like:

(15)



This tree is admissible only if GFA ignores stored categories in matching. It should also let any stored uncancelled categories passed up to the mother category. Before we reformulate GFA, we will further extend the notion of matching.

(16) **Extended Matching**
A. Category Matching
Let C and C' be categories, and {C} be a category C in storage. Then,
[i] C matches with C' if C' is a subcategory is C.
[ii] {C} also matches with C' if C matches with C'.
B. Sequential Matching

Let Z and Z' be sequences of categories of the length n, such as $Z=(C_1,\cdots,C_n)$ and $Z'=(C'_1,\cdots,C'_n)$.

[i] Z matches with Z' if every category C_i in Z matches with a unique category C'_i in Z'.

Furthermore, let $Z_{+i}{*}=(C_1,\cdots,C_i,\{C_i{*}\},C_{i+1},\cdots,C_n)$, and $Z'_{+j}{*}=(C'_1,\cdots,C'_j,\{C'_j{*}\},C'_{j+1},\cdots,C_n)$.

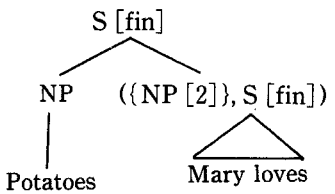
Then

[ii] $Z_{+i}{*}$ matches with Z' if Z matches with Z'.

[iii] Z matches with $Z'_{+j}{*}$ if Z matches with Z'.

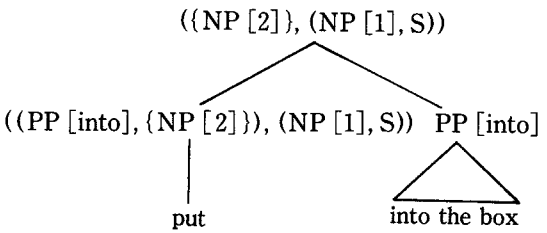
Here the definition of categorial matching given in (8) is extended to cover a case in which a stored category in a functor matches with its argument category and is thus cancelled. This, for instance, will be needed in dealing with topicalization.

(17)



The definition of Sequential Matching is also revised to accommodate cases involving sequences containing a stored category. This is needed to account for cases like:

(18)



Matching B[ii ,iii] simply says: "Ignore the occurrence of a stored category in matching."

Notice in (18) that the category PP[into] occurring in the sequence (PP [into], {NP[2]}) matches with the argument category PP[into] and is then cancelled but that the stored category {NP[2]} has no match and is left uncancelled. In general, we call such an uncancelled category a *remnant*. In B [ii , iii], stored categories $\{C_i{*}\}$ and $\{C'_j{*}\}$ are prospective remnants.

On the basis of the above notion of extended matching, we now show how

GFA operates to cancel out certain elements in a functor category.

(19) **Generalized Functional Application**
Let Z and Z' be any nonnull sequences of categories. And suppose Z matches with Z'. Then we have:
[i] GFA((Z,W),Z')=W'
[ii] GFA(({C}, (Z,W)),Z')=({C},W')
where C is a category in storage and W'=(R,W) if a remnant R results from Z's matching with Z'.

In GFA [ii], the occurrence of a stored category {C} is ignored in the operation of GFA.

For illustration, consider the category sequences in (14). As shown in (15), GFA[ii] applies to (14c) and (14d) where we have:

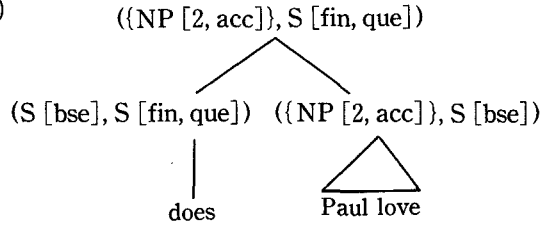
(20) a. {C}={NP[2,acc]}
b. Z=NP[1,nom]
c. W=S[bse]
d. Z'=NP[1,nom,s3,mas]

As a result, we obtain:

(21)
({C}, W')=({NP[2,acc]}, S[bse])
where there is no remnant from matching and thus W'=W

Next, GFA[i] can combine this result with an auxiliary verb *does,* admitting the following tree:

(22)



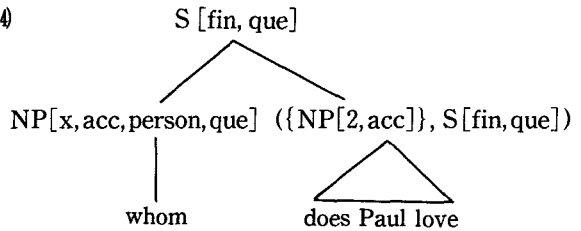
Here, the category of *does* is treated as a functor taking the category of *Paul loves* as argument so that when GFA applies to them we have:

(23) a. Z=S[bse]
b. W=S[fin,que]

c. Z'=({NP[2,acc]}, S[bse])
d. R={NP[2,acc]}
e. W'=({NP[2,acc]}, S[fin,que])

Finally, we obtain:

(24)



Here the stored category {NP[2,acc]} matches with the category NP[x, acc, person,que] of *whom* and thus is cancelled.

2.4. Syntax: Ordered Trees

Each GFA admits only an unordered local tree. This is a tree with a single mother immediately dominating daughter categories which are not linearly ordered to each other. Order is, however, introduced into each tree by observing the following linear precedence (LP) statements:

(1) **LP Statements**
[i] Functors precede their arguments.
[ii] But a functor of the category (A,S) is preceded by its argument category A unless S is marked with [inv(erted)].
[iii] Storage-fillers precede their corresponding storage categories.
[iv] Among sisters, we have: NP<PP<VP

We claim that the word order is language-particular. The above LP statements simply describe the phenomena of word order in English. Statement [i] makes a verb precede its complements and a determiner, its nominal expression. Statement [ii] allows the subject NP to precede its VP except for an inverted construction. Statement [iii] places WH-words, topic words, and relative clause antecedents at the beginning of clauses. Finally, statement [iv] fixes the order of verbal complements. We do not think that the above list is exhaustive. Stylistic variation plays an important role in ordering constituents of a sentence.

2.5. Lexicon: Agreement

Agreement in English is not a uniform phenomenon in English. The Subject

NP, for instance, agrees with its verb in number and person only. Determiners agree with thier nouns in number. In the case of a copular verb, the subject and the complement NP also agree in number. Object pronouns and WH-words agree with, or are governed by, their verbs with respect to case only. In order to account for such diverse phenomena in agreement, ACG simply adds a list of some pertinent agreement statements to the lexicon and builds appropriate pieces of information into some functor expressions in the lexicon.

Consider, first, the agreement between a verb and its subject as illustrated below:

(1)

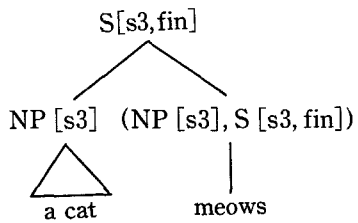
a. A cat meows.
b. *Cats meows.
c. *A cat meow.

(1a) is well-formed because the verb form of *meows* agrees with its Subject NP *a cat* in number and person, while (1b,c) are ill-formed because there is no such agreement there. To deal with this kind of agreement, we introduce the following principle into the lexicon:

(2) **Subject-Verb Agreement**
Given a functor category (NP,S),
(NP[⟨number,#⟩, ⟨person,%⟩], S[⟨number,#⟩, ⟨person,%⟩])

This statement simply says that NP and S in a functor category (NP,S) must agree in number and person. This then requires a verb like *meows* to belong to the category (NP[s3],S[s3,fin]). On the basis of such a lexical representation, GFA admits the following tree:

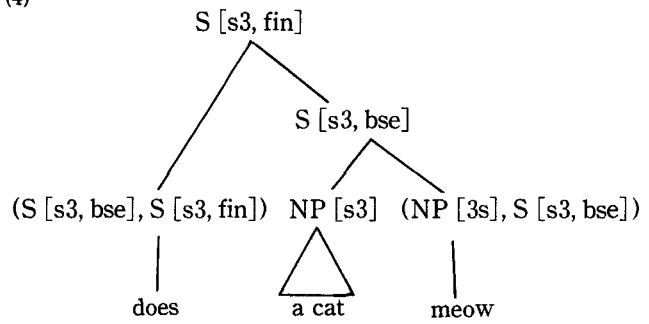
(3)



Note that Subj-V Agreement is not restricted to cases involving finite verbs only. It also applies to nonfinite verbs, thus allowing the category of nonfinite *meow* to be (NP[s3], S[s3,bse]). Suppose, furthermore, the auxiliary verb *does* belongs to the category (S[s3,bse], S[s3,fin]). Then by GFA we may obtain

the following tree:[5]

(4)

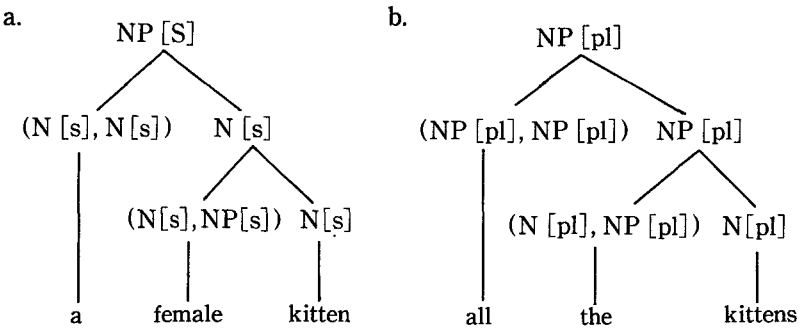


Next, we have a statement about the agreement in number between a determiner, an adjective, or any other adnominal expression and a nominal expression.

(5) **Adnom-N Agreement**
Given a category ([+N,−V], [+N,−V]),
we have: (+N,−V, ⟨number,#⟩], [+N,−V,⟨number,#⟩])[6]

This admits the following trees:

(6)



[5] The inflectional form of a verb varies only when it is finite. Hence, no inflectional change occurs in nonfinite verbs even if they have agreement feature specifications.

[6] Both N and NP belong to the category [+N,−V]. Note also that there is no distinction between N and N', or Nom, in our ACG, for they both have the same feature specification {-max}.

Note that our Adnom-N Agreement does more work than merely passing the number specification **s** or **pl** from the head N to the top node NP. The HFC in GPSG alone cannot account for this type of agreement. While Adnom-N Agreement blocks the following combinations, HFC alone cannot.

(7) a. *many kitten
b. *all (the) kitten[7]

Thirdly, in the case of copular verbs *be, become,* there is an agreement in number between the Subject and an NP complement. This again is captured by the following statement on agreement:

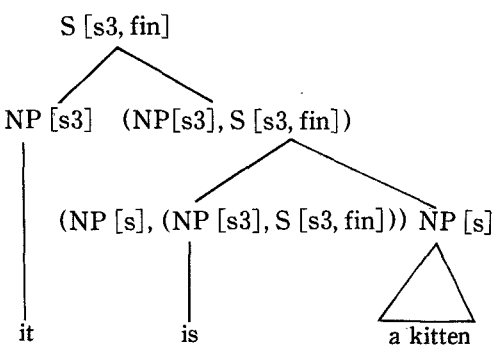
(8) **Copular Agreement**
If a copular verb belongs to a functor category (NP[⟨number,#⟩], (NP [⟨number,#'⟩], S)), then #=#'.

This statement, along with the statement on Subj-V agreement, will categorize the copula *is* as follows:

(9) is: (NP[s], (NP[s3], S[s3,fin]))

Hence, we will have the following tree:

(10)



Finally, pronouns are subcategorized with respect to gender, number, person, and case, and WH-words, with respect to case (and animacy). Hence, this information must be contained in the lexicon. We will, for instance, have:

[7] We do not know how to account for a well-formed expression like *many a kitten* at this stage.

(11) a. she: NP[s3,fem,nom]
b. her: NP[s3,fem,acc]
c. her: (N[#], NP[#])

(12) a. who: NP[nom]
b. who, whom: NP[acc]
c. whose: (N[#], NP[#])

Let us here be just concerned with case forms. Note first that the genitive forms *her* and *whose* are not treated as NP's, but of a functor category. So, in ACG, we have only two values for the feature **case**: **nom** and **acc**. Furthermore, if we take **acc** to be the default value for **case**, then we have to mark the value **nom** only.

Consider now how, when it combines with an NP, a verb selects an appropriate case form.

(13) a. She meows.
b. *Her meows.

(14) a. Paul loves her.
b. *Paul loves she.

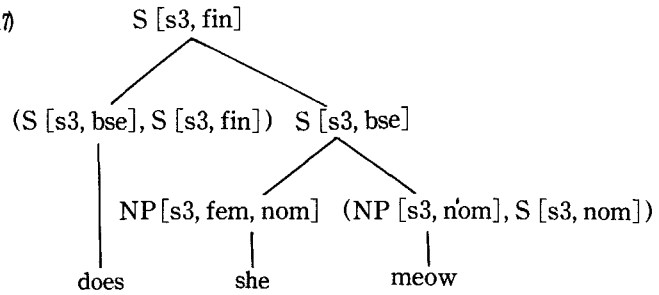
(15) a. Does she meow?
b. *Does her meow?

From this set of data, we simply note that Subject NP's are nominative case marked. Note in particular that this case marking is independent of the tense feature of verbs, for in (15) the Subject of a nonfinite, or base form, verb *meow* is assigned the nominative case. Assuming again that the default value of the feature **case** is **acc**, we introduce the following case restrictions into the lexicon:

(16) Case Assignment
[1] The default value of CASE is accusative.
[2] Given a category (NP,S), the value of the feature CASE for NP is nominative.

We may thus obtain the following tree:

(17)



Since the lexical item *meow* is of the category (NP,S), the CASE value of NP must be **nom**. Furthermore, NP and S here must agree in number and person. To be combined with *she* of the category NP[s3], their values must be **s3**. We thus obtain (17).

Consider now the following sentence:

(18) Who likes to meow?

To analyze this sentence, we should obtain the following information from the lexicon:

(19) a. who: NP[s3,nom,que]
b. likes: (VP[to], (NP[s3,nom], S[s3,fin]))
c. to: (VP[bse], VP[to])
d. meow: VP[bse]

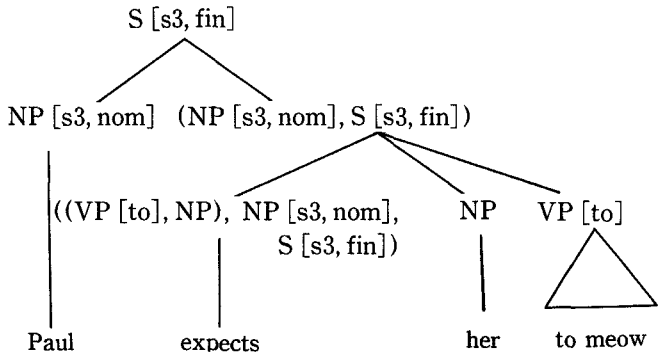
Note here that VP[bse] is an abbreviation of the category (NP[nom,#,%],S [#,%,bse]) and VP[to], that of (NP[nom, #,%], S[#,%,to]). The CASE value **nom** is assigned to each NP occurring in the category (NP,S) by the statement on Case Assignment. But since this functor is not combined with an NP in deriving sentence (18), this NP does not show up in the tree nor does it show any CASE assignment effect.

Consider then the following sentence in which the NP *her* is treated as the Subject of the complement VP *to meow*.

(20) a. Paul expects her to meow.
b. *Paul expects she to meow.

But in ACG this NP is the Object of the verb *expects* as shown in the following analysis tree:

(21)



Here, the NP *her* is treated as one of the two complements of the verb *expects*. But we will see later that *her* is interpreted as the "logical Subject" of the verb *meow* by a rule on control.

Finally, consider a case involving long-distance dependency:

(22) a. Who do you think meows?
b. *Whom do you think meows?

The variation in (22) can also be accounted for just seeing how the lexicon works. The lexicon as restricted by our statements on agreement will have the following feature specifications:

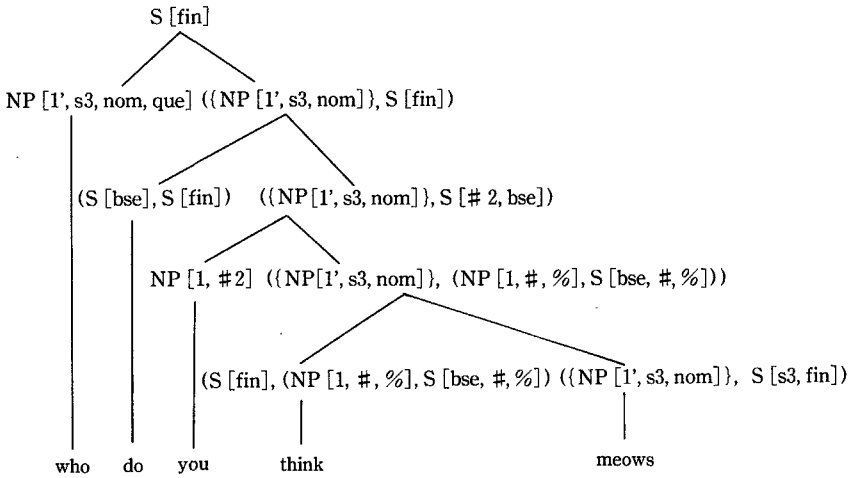
(23) a. do: (S[bse], S[fin])
b. you: NP[<NUMBER,#>, <PERSON,2>]
c. think: (S[fin], (NP[1,#,%,nom], S[#, %,bse]))
d. meows: (NP[1',s3,nom], S[s3,fin])

On the basis of (23d), we can obtain a storage category:

(24) meows: ({NP[1',s3,nom]}, S[s3,fin])

This then admits the following tree:

(25)



At the lexical level, the stored category is assigned **s3,nom** for its features NUMBER, PERSON, and CASE. This then is carried out to the higher nodes till the storage is filled by the question word *who* which has the same values for its features NUMBER, PERSON, and CASE.

As is illustrated, various agreement phenomena in English are accounted for by introducing some agreement statements about constructing the lexicon.

2.6. Syntax: Filling-in Storage

In the previous subsections, we have shown how we treat the following WH-questions:

(1) Who loves her?
(2) Whom does Paul love?
(3) Who do you think meows?

We obtain these sentences by treating WH-phrases as filler for storages. The verb *love* in (2), for instance, contains a storage in the Object position. This storage, or a gap, is filled in by its NP filler *whom.*

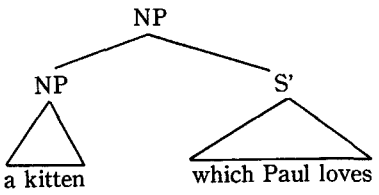
So-called gaps are also found in relative clause constructions in English. In this section, we will continue to show how storage categories are put to use for the analysis of these constructions with gaps. Consider the following.

(4) a kitten which Paul loves

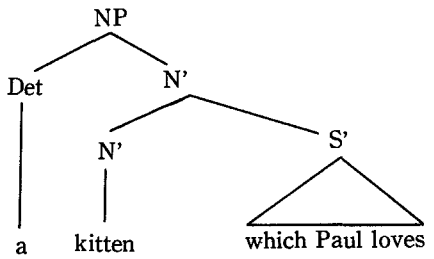
Here, the transitive verb *loves* is missing its Object. Through the bridging function of a relative pronoun *which,* the antecedent NP *a kitten* is interpreted as taking the role that is played by the Object.

In traditional literature, however, we find two syntactic analyses of this type of construction, the NP-S' analysis and the N'-S' analysis, as shown below:

(5) NP-S'Analysis



(6) N'-S' Analysis



Although classical Montague grammar and GPSG have adopted the N'-S' analysis which seems to render a simpler semantic description, we will show here how the NP-S' analysis as commonly accepted in TG or GB can easily be accommodated into our ACG.

Unlike WH-question phrases, we treat relative pronouns as phrases of a functor category as schematized below:

(7) COMPrel=(({NP[x,#,%,@]}, S[fin]), ({NP[x,#,%,@']}, NP[y,#,%, @']))
where x, y are parameters of INDEX,
that of NUMBER,
% that of PERSON, and
@, @' those of CASE.

This then can be instantiated into various forms:

(8)

a. which: (({NP[x, #, %, @]}, S[fin]), ({NP[x, #, %, @']}, NP[y, #, %, @']))

b. who: (({NP[x, #, %, nom]}, S[fin]), ({NP[x, #, %, @]}, NP[y, #, %, @]))

c. whom: (({NP[x, #, %, acc]}, S[fin]), ({NP[x, #, %, @]}, NP[y, #, %, @]))

It should be noted that these parameters must be anchored to specific values whenever possible and that the occurrence of a parameter in a category may create a crucial difference. For example, unlike a simple NP, the occurrence of an INDEX parameter x in NP[x] makes this NP an indexed NP and thus a candidate for storage.

On the basis of (8), we can easily derive the following relative NP's:

(9)

a. kittens which Paul loves

b. Paul, who loves Pommy

c. a girl who we think meows

For convenience's sake, we shall represent each analysis in a vertical form by first listing all the lexical signs and then each phrase obtained by GFA. We also annotate the anchoring of parameters and the specification of other feature values forced by GFA under each appropriate constituent category.

By analyzing (9a), we first show that the antecedent *kittens* has the same INDEX value as that of the stored category in the Object position of the verb *loves*, thus being interpreted as playing the role of being loved by Paul.

(10)

a. a kitten: NP[z, s3]/z=2, @'

b. which: (({NP[x, #, %, @]}, S[fin]), ({NP[x, #, @']}, NP[y, #, %, @']))/x=2, @=acc

c. Paul: NP[x, s3]/x=1, nom

d. loves: ({NP[2,acc]}, (NP[1,s3,nom], S[fin, s3]))

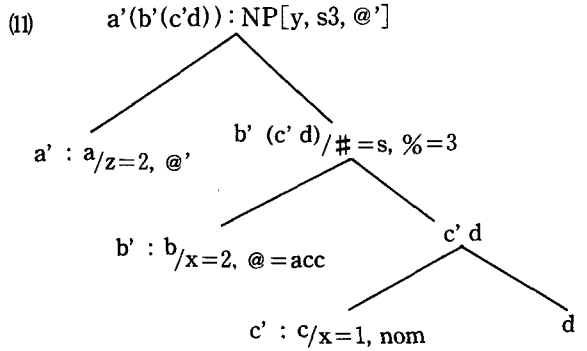
cd. Paul loves: ({NP[2,acc]}, S[fin, s3])

b(cd). which Paul loves: ({NP[2, #, %, @']}, NP[y, #, %, @'])/#=s, %=3

a(b(cd)). a kitten which Paul loves: NP[y, s3, @']

(cd) is obtained by first anchoring x to 1 and forcing the CASE **nom** to (c) and then cancelling NP[1, s3, nom] in (d). (b(cd)) is then obtained by a similar process. Here, the INDEX parameter x is anchored to 2 and the CASE parameter, to **acc**. Finally, (a(b(cd)) is obtained with the INDEX parameter z in (a) being anchored to 2 and with the NUMBER and CASE parameters #

and % in (b(cd)) anchored to s and 3, respectively. This whole process of anchoring, forcing, matching and cancellation under GFA may be represented by the following annotated tree:



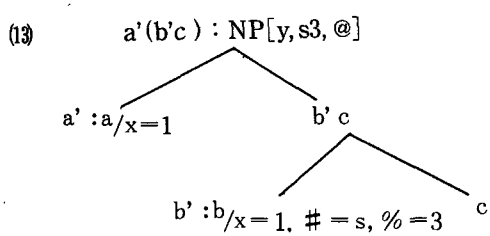
On the basis of lexical signs (a), (b), (c), and (d), we can interpret the above tree and describe how the category at the topmost node is obtained. This singular 3rd person NP has two latent parameters y and @′ for ROLE INDEX and CASE, which will be determined when it is used in a large context.

As claimed by Cooper and Engdahl (in preparation), we don't distinguish so-called nonrestrictive relative clauses from restricive ones. We thus treat (9b) just like any other relative clauses. Here we show that the antecedent *Paul* has the same ROLE INDEX as the stored Subject of a verb phrase *loves Paul* and also that they agree in NUMBER and PERSON. For (9b), we assume to have been given the following piece of basic information:

(12)

a. Paul: NP[x, s3]
b. who: ({NP[x, #, %, nom]}, S[fin]), ({NP [x, #, %, @]},NP[y, #, %, @])}
c. loves Pommy: ({NP[1, s3, nom]}, S[fin, s3])

On the basis of (12), we then construct the following annotated tree:



Note here that the relative pronoun *who* functions as a bridge for relaying information concerning ROLE, NUMBER, and PERSON between the antecedent and the storage or the gap. Note also that the CASE value of the relative pronoun is determined by the CASE value of its corresponding storage.

Finally, we analyze (9c) to show how the storage in an embedded sentence may be filled by its antecedent through a bridging relative pronoun COMPrel. We are again provided with some basic pieces of information concerning each lexical item of phrase occurring in (9c).

(14)
a. a girl: NP[x,s3]
b. who: (({NP[x, #, @, nom]}, S[fin]), ({NP[x, #, %, @]}, NP[y, #, %, @]))
c. we: NP [x, pll, nom]
d. think: (S[fin], (NP[1, #, %, nom], S[#, %, fin]))
e. meows: ({NP[1', s3, nom]}, S [s3, fin])

GFA applies to (d) and (e), yielding:

(15) $(d_{/S3})$ e think meows: ({NP[1', s3, nom]}, (NP[1, #, %, nom], S[#, %, fin]))

By anchoring the ROLE INDEX parameter x in (c) to 1 and also the NUMBER and PERSON parameter # and % in $((d_{/S3})e)$ to pl and 1, respectively, we obtain:

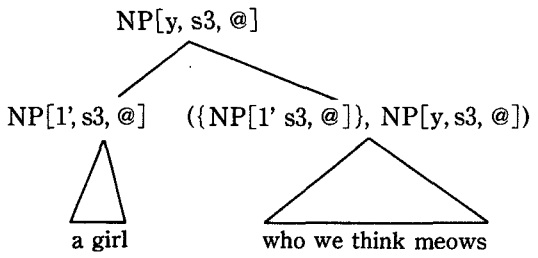
(16) $((c_{/x=1})((d_{/S3})e)_{/\#=pl,\ \%=1})$ we think meows: ({NP[1', s3, nom]}, S[pl, 1, fin])

This then combines with $(b_{/x=1',\ \#=s,\ \%=3})$, yielding:

(17) who we think meows: ({NP[1', s3, @]}, NP[y, s3, @])

Finally, we anchor the INDEX x in (a) to 1' and obtain (9c). The topmost local tree is represented as in:

(18)



This tree is fully admissible by our generalized functional application GFA.

2.7. Grammatical Functions and Semantic Roles

In ACG, every NP is to be assigned a grammatical relation GR. This GR is purely a syntactic notion such that even so-called dummy NP's, or expletives, are assigned a GR.

We have two GR's, Subject and Object, in the grammar. These notions are defined as follows:

(1) Subject
The NP occurring in the functor category (NP,S) is the Subject of S.

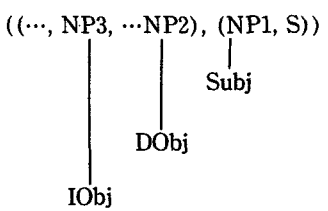
(2) Object
Let C and C' be any categories such that C' = ((..., NP_i...),C). Then each NP_i occurring in C' is an Object of the category C'.

Suppose we have a category, (NP2,(NP1,S)). Then, NP1 is the Subject of S, and NP2 the Object of the entire category, that is, a verb. For another example, consider a preposition of the category (NP, PP). This NP is a prepositional Object.

Some categories may have multiple Objects. In such a case, these Objects are distinguished by their position in the sequence. By convention, we may name the rightmost NP in the category sequence (...,NP_i,...) within a functor category ((...,NP_i,...),C) Direct Object and the second rightmost NP Indirect Object.

Our definition of grammatical relations can be illustrated as below:

(3)



Theoretically speaking, there may be some non-NP categories intervening between the Direct and the Indirect Object. So we allowed dots between NP3 and NP2 above.

Grammatical relations are, in a proper sense, relations holding between functors and their argument categories. But we have defined them as relations on constituent categories of a functor category. This is certainly a departure from the traditional way of defining grammatical categories. It will not, however, result in confusion as long as we understand GR as an attribute of a category, for every such information encoded in it is transferred to its argument category by the convention of forcing on category matching. As a consequence, GR's are eventually assigned to argument categories.

Unlike GR's, semantic roles are not assigned to each NP in a sentence. Specifically, no semantic role is assigned to expletives. The impersonal pronoun *it* is not assigned a role, either. Excepting these few, NP's are in general potential role carriers.

But strictly speaking, roles are not assigned to linguistic or metalinguistic objects like expressions and categories, but to real objects that are involved in some relation. Consider a statement made by the following sentence:

(4) Paul loves Pommy.

This statement may be interpreted as describing a situation in which Paul loves Pommy. Here it is not the NP's occurring in sentence (4), but the boy named Paul and the female kitten named Pommy referred to by those NP's that are understood as each having a unique role in the relation of love. Only in modelling such a situation and talking about such roles, we can say in general that, given a basic state of affairs of the type $\langle\langle R, a_1 ..., a_n ; 1\rangle\rangle$, we can say that each object assigned to each $\mathbf{a}_i$ plays a unique role with respect to the relation **R**. Abstracting from real situations one step further away, we may also be able to speak of roles as if they were associated with certain categories of expressions if they occur in a certain grammatical construction.

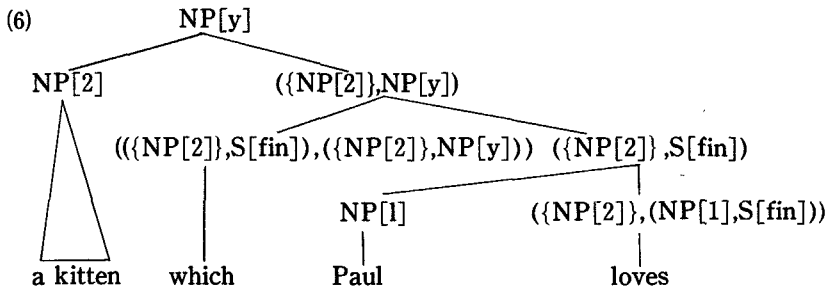
The feature INDEX is thus introduced to make it possible to talk about "semantic" roles of categories. The transitive verb *love*, for instance, belongs to a functor category (NP[2], (NP[1],S) with each NP occurring in it assigned an INDEX value. In the semantic part of its lexical sign, we have as its meaning a state of affairs $\langle\langle$LOVE, IND.1, IND.2 ;1$\rangle\rangle$. We then interpret NP [1] and NP[2] as each taking the role IND.1 and IND.2, respectively, in the relation of love.

Consider the case of a lexical sign *kitten* where a parameter is assigned as an INDEX value:

(5) kitten N[x]
|
<<KITTEN,IND.x ;1>>

From this sign, we are to understand that an instance of the parameter x in N [x] denotes an individual that has the role of being a kitten.

Consider a bit more complicated case:



From this tree, we are supposed to be able to talk about the semantic roles of *a kitten* NP[2] of the entire phrase *a kitten which Paul loves* NP[y] if we know about the states of affairs in which the objects denoted by these NP expressions play roles. In the case of NP[2], we understand that the individual denoted by *a kitten* of NP[2] plays the role of being loved. But in the latter case, we only know that the individual denoted by the entire phrase NP[y] has some potential role associated with the parameter y.

So far we have just appealed to intuitive understanding in explaining how the INDEX feature can be used to interrelate SYNTAX with SEMANTICS and to obtain the information about semantic roles. The use of INDEX will become more apparent if we show how a parameter is anchored to an object and how states of affairs are merged in our ACG system.

2.8. Semantics: Merging

By merging various states of affairs together, we accumulate picecs of information conveyed by each of them. For example, given the following two basic states of affairs, or simply soa's,

(1)
a. <<KITTEN, x ;1>>
b. <<IND,x ; 1>>

we can merge them into a larger state of affairs in which some individual object x is a kitten.

(2)

<<KITTEN,x ; 1>>
<<IND,x ; 1>>
OR simply, <<KITTEN, IND.x ; 1>>[8]

This soa is then interpreted as conveying the information about some x accumulated from pieces of information about the same x conveyed by soa's (1a) and (1b).[9]

Consider another example :

(3)

a. <<KITTEN, IND.x ; 1>>
b. <<FEMALE,x ; 1>>

Merging soa's, in general, means putting them together. From (3), we obtain :

(4)

<<KITTEN, IND.x ; 1>>
<<FEMALE,x ; 1>>

This merged state of affairs is interpreted as conveying the accumulated information about x that it is an individual female kitten.

But sometimes merging means more than just putting soa's together. It requires setting the value of some index to a parametar or an object. consider the following example :

(5)

a. <<=,IND.x,Pommy ; 1>>
b. <<LOVE, IND.1,IND.2 ; 1>>

Here nothing particular happens unless either of the indices 1 or 2 is assigned an appropriate value. Suppose we set the value of 2 to the parameter x. Then from the two soa's we obtain the following merged soa :

(6)

<<=,IND.x,Pommy : 1>>
<<LOVE, IND.1,IND.x ; 1>>

[8] IND x may be interpreted as "the parameter x restricted to the type IND OF individual objects.

[9] A parametric soa, that is, a soa containing a parameter, is fully interpreted only when each parameter occurring in it is anchored to an appropriate value, or a real object.

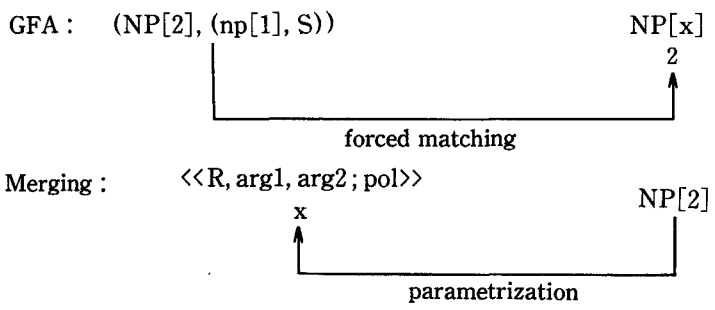
On the basis of **Substitution for Identicals**, this merged soa can be shown to convey the same information as:

(7) <<LOVE,IND.1,IND.Pommy ; 1>>

The above soa contains the information about Pommy that it is loved. Here again the merging of soa's has resulted in the accumulation of information about Pommy.

The setting of INDEX values to parameters is constrained by the syntactic operation GFA. If GFA has forced the value of the parameter x in NP[x] *Pommy* to be set to 2 in order successfully to concatenate it with an expression *loves* of the category (NP[2], (NP[1],S)), then that value of INDEX, namely 2, which is labelled to one of the argument roles in the soa described by the verb *loves* should be replaced by the parameter of the parametric soa described by the NP *Pommy* in merging these two soa's. This operation can be seen as **de-labelling**.

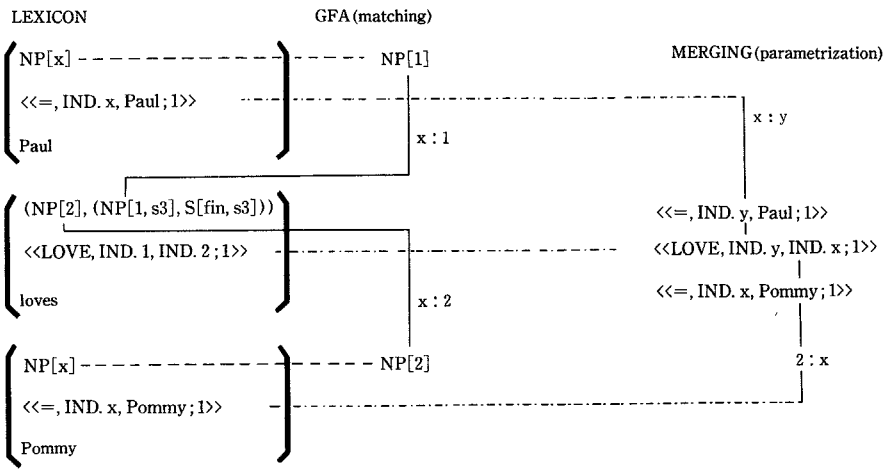
(8)



Corresponding to the syntactic category Co of a functor expression is a soa labelled with argument roles. If GFA forces the parameter of its argument category Ci to be matched with any of the INDEX values in Co, then MERGING replaces the argument role corresponding to the argument category Ci with an appropriate parameter.

The following illustrates how GFA and MERGING interacts with each other:

(9)



As shown above, parametrization in MERGING not only sets the value of an index to a parameter, but also replaces a parameter with another parameter to avoid the collision of parameters.

Parametrization assigns a distinct value unless specified otherwise. In a case, for instance, involving reflexive pronouns, we require two distinct indices corresponding to two distinct argument roles to be replaced with one and the same parameter.

(10)

a. Pommy scratches herself.
b. <<SCRATCH,IND. x, IND. x; 1>>
<<=,IND.x,Pommy; 1>>

In (10), Pommy plays the role of a scratcher and at the same time that of a scratchee.

3. Implementation of GFA In Prolog

In this section, we present the implementation strategy of GFA in Prolog. Before presenting the entire program, the representation schemes for each element used in ACG will be discussed.

3.1. Brief Introduction to Prolog

Prolog is a logic programming language which is well suitable for the

language parsing. Since it is a kind of relational language, it is very easy to implement the functional application in it. The Prolog program consists of facts and rules. Facts are instantiated predicates, i.e., propositions representing facts which holds in the domain. Rules are statements of the form:

A :-B1, B2,···,Bn.

, where $n \geqq 0$. A is the head of the rule, and the Bi's are its body. It has both the declarative and procedural semantics. As a declarative meaning, it is interpreted as that A is true if B1, B2,···, and Bn are true. It is interpreted procedurally as follows:

To prove A, we must prove B1, B2,···, and Bn.

The logic program is a finite set of rules and each rule is a Horn clause.

3.2. Representing Categories

The syntactic category of ACG is represented by the complex term in Prolog. It means that each category is denoted by the functor symbol[10] and a set of features is represented by the argument list of the complex term as follows:

Category([F1,F2,···,Fn])

, where each Fi means the feature of the category *Category*. For example, the syntactic category of NP *Paul* is represented as follows:

np([X,s3,masc])

This representation denotes that the category of Paul is that it is a 3rd person singular masculine NP. Here, X corresponds to the index parameter.

A functor category is represented as a pair of categories. Consider the case of transitive verb *loves*. It has the syntactic component as below:

(NP[2,acc], (NP[1,s3,nom],S[fin]))

It is implemented as the following representation:

(np([Y,acc]),(np[X,s3,nom], s([fin])))

Here, we use the logical variable X, Y to specify the index parameters for the nominative NP and accusative NP, respectively. Since it is denoted as different logical variables, they are instantiated by the separate substitution. Even

[10] Don't confuse the functor symbol in Prolog with the functor category in ACG.

though they are represented by the different variables, they can be anchored to the same individual when X and Y are unified by other conditions. This is reasonable because the accusative NP and nominative NP can be the same person.

The representation of the storage category is quite similar to the other categories except that the qualified bracket { } is used in its representation. For example, the syntactical component of a relative pronoun *who* appeared in (14) of Section 2.2 which has a storage, is represented as below:

(({np([X,nom,person,rel])}, s([])),(np([X]),np([X])))

, where the component between {and} is a stored category.

3.3. Use of Logical Variables As Index Parameters

For representing the index parameters in ACG, we use the logical variables. When a specific individual is matched, the logical variable is instantiated by that individual via the unification provided in Prolog. For example, consider the derivation tree (2) in Section 2.3. In this tree, the index parameter of NP *Pommy* is set to 2 in the functional application. This is done by the functor category of the verb *loves*. This can be accomplished in our system by unifying two variables corresponding to the index parameter of the category of Pommy and the index parameter of the argument category of *loves*. In Prolog, the unified variables are thought to denote the same constant and, in fact, when a variable is instantiated to a specific constant, all unified variables are also instantiated to the same constant automatically.

By using the logical variable as index parameter, we can simplify the matching and anchoring of index parameter. Actually, the notation itself used in the index parameter is meaningless. They are used only in GFA procedure for matching. Thus, by the logical variables and the unification in Prolog, we can easily combine the parameters so that they point appropriate individuals.

The logical variable makes it easy to unify the semantic component by the use of the same variable in the semantic representation. As we said before, when a variable is instantiated to a specific individual, all the same variables and unified variables are instantiated to that one simultaneously. Therefore, as the nominative and accusative NPs in a verb *loves* are anchored to individuals, their corresponding variables in the semantics of *loves* are also instantiated to the same individuals since they are represented by the same variables, respectively.

Consider the derivation tree (2) in Section 2.3 once again. In this derivation, as the index parameter for each category is matched, the parameters in the semantic representation of the verb *loves* are instantiated to Paul and Pommy, respectively. That is, as Pommy is used as an accusative NP in this derivation,

it is substituted as an individual of IND.2 by semantic unification. Here, we do not describe the semantic unification precisely. We hope that it will be presented in a later paper.

3.4. Implementation of GFA

Prolog has both the declarative and procedural semantics. In addition, since it is a kind of relational language, it is very easy to implement the GFA procedure according to its functional semantics. By the definition of GFA as shown in Section 2.5 (20), the top level procedure **gfa** for GFA is given as follows:

```
gfa(({Storage}, (CAT1, CAT2)), ARGSEQ, ({Storage}, CAT2)) :-
  match_cat(CAT1,ARGSEQ).
gfa((CAT1,CAT2), ARGSEQ,CAT2) :-
  match_cat(CAT1,ARGSEQ).
```

In this program, the first rule shows the case when the storage category is included[11]. The identifiers starting with the upper case letter are variables in Prolog. The unification scheme used in Prolog allows a variable to be a sequence of categories. The third argument is a return value of the GFA procedure. The **match cat** predicate is satisfied if the first argument matches with the second argument **ARGSEQ** which is a sequence of the argument categories. The second rule shows the normal case of GFA.

In order for a category to be matched with other category, the latter should be a subcategory of the former. It is checked by the feature list of category. In other words, A' is a subcategory of A if A is a subset of A' by definition (7) in Section 2.5. It can be proved by comparison of each element of two feature lists. The following program shows the check of the subcategory relationship between two categories[12].

```
sub_cat(C1,C2) :-
  C1 =.. [CAT, Feature 1],
  C2 =.. [CAT, Feature 2],
  sublist(Feature 1,Feature 2).

sublist ([], _).
sublist ([H | T], [H | T2]) :-
  sublist (T,T2)
```

The first and second subgoal in the first rule **sub_cat** are to separate the argument of the complex terms **C1** and **C2** representing categories, respective-

[11] The first rule of gfa corresponds to (19ii) of §2.3.

[12] The second rule of gfa corresponds to (19i) of §2.3.

ly. The feature list of each category is denoted as **Feature1** and **Feature 2**, respectively. After that, the procedure **sublist** checks if the first list contains less elements than the second one. Of course, the common elements should be equal to each other. The first statement of the procedure **sublist** means that the number of elements of the second argument is greater than or equal to that of the first, for the first argument is the empty list []. The anonymous variable (_) denotes any variable. The second statement means that if the first elements **H** of each list are the same, then the remaining elements **T** and **T2** are checked in the subgoal **sublist (T,T2)**.

The whole program is contained as Appendix of this paper. Followings are some examples of calling of the GFA procedure.

```
?- gfa((np([Y,acc]),(np([X,s3,nom]),s([fin]))), np([Z,acc]),VAL).

VAL=(np([_91,s3,nom]),s([fin]))
yes
?- gfa((np([Y,acc]),(np([X,nom]),s([fin]))),np([2,acc,s3,person,fem]), VAL).

VAL=(np([_91,nom]),s([fin]))
yes
?- gfa(((pp([in]),np([Y,acc])),(np([X,nom]),s([]))),(np([2,acc,s3,impers]),
pp([in])),VAL).

VAL=(np([_102,nom]),s([]))
yes
?- gfa(({np([Y,acc])},(np([X,nom]),s([bse]))),np([Z,nom]),VAL).

VAL=({np([_82,acc])},s([bse]))
yes
```

In this sample session, the notation starting with an underbar (_) means the internal representation of variables in Prolog, like **_91**. This program **gfa** is used in the paring phase of natural language, based on ACG.

4. Conclusion

Our description of ACG has been sketchy and its implementation, partial. In this paper, we have been mainly concerned with spelling out the definition of categories and the notion of generalized functional application. This much, we believe, has been accomplished here.

ACG differs from classical categorial grammar in two main respects. First,

as in GPSG, it treats a basic category not as an unanalyzable atom, but a set of feature-value pairs. Note, however, that ACG does not introduce category-valued features, but only atom-valued ones. Secondly, it introduces storage categories to deal with gaps (for instance, involving WH-questions). On the basis of these categories, functional application is generalized. Our notion of storage categories is preferred to that of stacks because ours perserves the information as to where categories are stored or where gaps are possible.

ACG resembles HPSG in many respects. But unlike HPSG, ACG does not rely on the notion of head, but rather on that of functor. A functor is treated in ACG as a pivotal category because it contains a full piece of information concerning the local configuration of a well-formed phrase. Given a functor, we know exactly what kind of tree can be obtained from it, for it provides the information concerning the mother and its daughter categories.

GFA is simply a reformulation of functional application based on the new definition of categories. It especially accommodates storage categories and treats stored categories as reserved for use at the final stage of functional application. Stored categories are thus ignored in the operation of functional application till they are absolutely needed.

We have shown in this paper how these notions of categories and GFA apply to agreement and WH-gap phenomena in English. For agreement, we have introduced into ACG something that resembles feature cooccurrence restrictions in GPSG. And for WH-constructions, we utilize storage categories.

Our implementation of ACG in Prolog has been restricted to categories and GFA. We have not shown how it works for those cases illustrated in section 2. But it should be a routine application of our computational formulations.

We have claimed that the semantics of ACG is situation-theoretic or information-based. Section 2.8 has been written to indicate our interest, but not too adequately developed to support our claim. This we hope to do in our future work.

REFERENCES

Barwise, Jon and John Perry (1983) *Situations and Attitudes,* The MIT Press, Cambridge, Mass.

Clocksin, William F. and Christopher S. Mellish (1985) *Programming in Prolog,* 2nd ed., Springer-Verlag, Berlin.

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum and Ivan Sag (1985) *Generalized Phrase Structure Grammar,* Basil Blackwell, Oxford.

Pereira, Fernando C. N. and David H. D. Warren (1980) 'Definite Clause Grammars for Language Analysis -A Survey of the Formalism and a Comparison with Augmented Transition Networks,' *Artificial Intelligence* 13, 231-278, North-Holland Publishing Company, Amsterdam.

Pollard, Carl (1985) *Lectures on HPSG,* Unpublished.

Sag, Ivan and Carl Pollard (in preparation) *An Informatioin-Based Approach to Syntax and Semantics.*

APPENDIX

```
/*****************************************************************/
/* Generalized Functional Application                            */
/* o First Argument denotes the Functor Category.                */
/* o Second Arguments are a Sequence of Argument Categories.     */
/* o Categories are represented as complex terms.                */
/* o{···} denotes the Storage.                                   */
/*****************************************************************/

gfa(({STG}, (CAT1, CAT2)),ARGSEQ,({STG}, CAT2)) :- % If contains
  storage
  match_cat(CAT1,ARGSEQ).
gfa((CAT1,CAT2),ARGSEQ,CAT2) :-
  match_cat(CAT1,ARGSEQ).

/* Checking if the first argument is matched to the second*/
match_cat((CAT1,T), ARGSEQ) :- % if the first one is a sequence
  match(CAT1,ARGSEQ),
  match(T,ARGSEQ).
match_cat(CATSEQ,ARGSEQ) :-
  match(CATSEQ,ARGSEQ).

/*Procedure for determining if the second sequence contains
  the subcategory of the first category*/
match(CAT,(CAT1,T)) :-
  sub_cat(CAT,CAT1).
match(CAT,(CAT1,T)) :-
  match(CAT,T).
match(CAT,CAT1) :- % CAT1 is not s sequence
  sub_cat(CAT,CAT1).
/*Procedure for checking if the second argument is a
  subcategory of the first */
```

```
sub_cat(C1,C 2) :-
  C1 = · · [CAT,Feature 1],
  C2 = · · [CAT,Feature 2],
  /*If the same category, check the feature list */
  subfeature(Feature 1,Feature 2).

/*Subfeature satisfies if the first list contains less number of same ele-
ments than the second list */
subfeature([],_).
subfeature([H | T], [H | T2]) :-
  subfeature(T,T2).
```

(Kiyong Lee, Suson Yoo)
Department of English
Korea University
1 Anam-dong, Seongbuk-ku
Seoul 132
Korea

(Key-Sun Choi, Sangki Han)
Department of Computer Science
Korea Advanced Institute of Science and Technology
P.O.Box 150, Cheongryang
Seoul
Korea