LOGIC GRAMMAR TOOLS FOR COMPUTATIONAL LINGUISTICS

Harvey Abramson
Institute of Industrial Science, University of Tokyo
Minato-ku, Roppongi 7-22-1
Tokyo 106, Japan
e-mail harvey@godzilla.iis.u-tokyo.ac.jp

Abstract. To many people, logic grammars are equivalent to the Definite Clause Grammars (DCGs) of Warren and Pereira. Although sophisticated applications can be built with DCGs, the notation is too close to Prolog and too far from linguistic notation. Furthermore, DCGs rely entirely on a very simple control stategy for execution. In this paper we describe some advances in notation and control which make logic grammars more declarative and which bring logic grammars closer to current linguistic concerns. In particular, we shall describe the addition of feature sets and constraints on feature values, delaying mechanisms to implement such constraints declaratively, metaprogramming for logic grammars, and the application of metagrammatical programming to ID/LP parsing.

The logic grammar background. Derivations in logic programs are, in a sense, a generalization of context free derivations. Consider a conjunction of goals in a logic program:

$$:- G_1,G_2,...,G_i,...,G_n$$

The next derivation step yields the following conjunction of goals

$$:- (G_1,G_2,...,G_{i-1},B_1,...,B_m,G_{i+1},...,G_n)\theta$$

provided that there exists a clause in the program

$$G :- B_1,...,B_m$$

such that G unifies with G_i under the unifying substitution θ. In case there is a fact or unit clause G which unifies with G_i, the next derivation step becomes:

$$:- (G_1,G_2,...,G_{i-1},G_{i+1},...,G_n)\theta$$

The derivation may terminate with the empty clause (empty conjunction of goals), signifying success, and then the unifying substitutions provide values for any variables occurring in the initial goal clause; or, it may terminate with a non-empty conjunction of goals, indicating failure of the original goal. Unfortunately, the derivation process need not terminate.

Now consider derivations using a context free grammar. Given a sentential form containing at least one nonterminal symbol A

$$a_1...a_iAa_{i+1}...a_m$$

the next step in the derivation is

$$a_1...a_ib_1...b_ma_{i+1}...a_m$$

provided that there exists a context free production

$$A \text{ --> } b_1...b_m$$

in the grammar. The derivation process terminates when a sentential form contains only terminal symbols.

A context free grammar can be easily represented as a logic program and context free derivations can then be subsumed by the process of evaluation of a logic program goal. A grammar rule such as

$$A \text{ --> } b_1...b_m$$

is represented by the logic program clause

$$a(X_0,X_m) :- b_1(X_0,X_1),...b_m(X_{m-1},X_m)$$

If b_i is a nonterminal there is a logic program clause for each grammar rule with b_i in the left hand side. If b_i is a terminal symbol t_i then

$$b_i(X_{i-1},X_i) = \text{connect}(t_i,X_{i-1},X_i)$$

where the unit clause connect is defined by

connect(x,[x|X],X).

The arguments $X_0...X_m$ represent the source language string considered as a "difference list". Thus, the logic program clause above is read as "There exists an a between X_0 and X_m provided that there is a b_1 between X_0 and X_1, ..., and a b_m between X_{m-1} and X_m. The connect predicate succeeds when the first component of a difference list matches the sought for terminal symbol x. If sentence is the start symbol of the context free grammar, then the following goal will analyse or generate a sentence X of the language

:-sentence(X,[])

Definite Clause Grammars (DCGs - see [16]) generalize the above mapping by allowing each predicate b_i to have further arguments in addition to X_{i-1} and X_i. These extra argument places can be used to enforce various linguistic constraints, build a derivation tree structure, or compute semantic representations. Here is a Definite Clause Grammar

which appears in [5] which makes use of an extra argument to specify when a direct object may be elided, and which also prevents a direct object from being elided when it should not be. The value of the argument is 'not_elided' when the object should not be elided, and 'elided' when it is. Terminal symbols are written enclosed within square brackets, such as [the]. In the right hand side, [] represents the empty string.

```
(0)  sentence --> sent(not_elided).
(1)  sent(E) --> noun_phrase,verb_phrase(E).

(2)  noun_phrase --> determiner,noun,relative.
(3)  noun_phrase --> proper_name.

(4)  verb_phrase(not_elided) --> verb.
(5)  verb_phrase(E) --> transitive_verb, direct_object(E).

(6)  relative --> [].
(6') relative --> relative_pronoun,sent(E).

(7)  direct_object(not_elided) --> noun_phrase.
(7') direct_object(elided) --> [].

(8)  determiner --> [the].

(9)  noun --> [man].

(10) proper_name --> [john].

(11) verb --> [laughed].

(12) trans_verb --> [saw].
(13) relative_pronoun --> [that].
```

For this grammar, the following query will succeed.

```
:- sentence([the,man,that,john,saw,laughed], []).
```

Furthermore, by enclosing any conjunction of logic programming goals within a pair of braces in the right hand side of a DCG rule, other constraints, not expressible as ordinary grammar rules, may be specified.

$$a(X_0,X_m) :- b_1(X_0,X_1),...,\{Goals\},...,b_m(X_{m-1},X_m)$$

The mapping from DCG rules to logic programming predicates is a simple mechanical process (essentially adding the arguments which hold the representation of the source string to nonterminal symbols and generating calls to connect for terminal symbols) so that the DCG user simply writes DCG rules which automatically get translated into logic program clauses. The usual Prolog implementation of DCGs relies on Prolog's left to right, depth first control strategy to evaluate such goals as sentence(X,[]), and therefore the parsing method is essentially recursive descent, and very often generation cannot be done as easily as analysis.

done as easily as analysis.

Since the introduction of DCGs, several other logic grammar formalisms have been introduced which attempt to generalize DCGs by dealing automatically with one linguistic problem or another. Usually, the generalization is handled by mapping grammar rules into logic program clauses with several automatically added arguments in addition to those which hold the input/output string as a difference list. A case in point is Pereira's Extraposition Grammars (XGs see [17]) in which the logic program clauses contained a total of four automatically generated arguments: two for the difference list, and two to manipulate a list of possibly left-extraposed elements. We will not go into detail here about this formalism, but the reader may consult [5] for a discussion of XGs and other such formalisms.

Although the basic idea introduced by DCGs is one which permits the development of serious computational linguistic applications, the process of doing so by using arguments attached to nonterminals introduces problems of software engineering and what might be called programming style. In any complex application, large numbers of extra arguments may be needed, and then the difficulties of keeping track of which argument represents which linguistic problem become an enormous obstacle to the maintenance and development of the system. Further, there has been a tendency in the field of logic grammars to introduce a new formalism whenever some linguistic problem could be hidden away in arguments automatically attached to logic program clauses. In the rest of this paper we shall discuss how these problems of developing logic grammar applications may be simplified. The former problem can be dealt with by making use of a system of feature sets and an extended notion of unification derived from unification grammars. This idea is not new, of course, (it dates back to early attempts to implement, for example, Lexical Functional Grammars), but it is one which can easily be adapted to the logic grammar framework. The second problem can be dealt with by adapting the logic programming notion of metaprogramming to extend a basic logic grammar formalism. In the rest of the paper we shall discuss these changes. In particular, the feature system which we will introduce will also make use of some advanced logic programming control techniques to gain an added measure of declarativity.

Feature Sets and Extended Unification. Over the last few years the so-called unification grammars have become popular among linguists and computational linguists. One aspect of these grammars is the representation of grammatical information, syntactic, semantic, pragmatic, etc. in terms of sets of features. A feature set is essentially a collection of name-value pairs in which the value itself may be a feature set or a variable or an atom (a name, essentially).
For example:

```
{ category = noun,
  lexical  = cat,
  type     = {animate = yes,
              human   = no}
}
```

represents some of the information associated with the word "cat". Further information may be contained in another feature structure:

```
{ lexical = cat,
  category = N,
  type     = X,
  number = singular,
  species = 'Siamese'
}
```

The information contained in these structures may be combined by a general notion of unification to yield:

```
{ category = noun,
  lexical = cat,
  type = {animate = yes,
          human = no},
  number = singular,
  species = 'Siamese'
}
```

If a name appears at the same level of both feature sets, then that name appears in the result providing that the values associated with that name unify (in the general sense); the value associated with the name in the result is the unification of the associated values of the original feature sets. Otherwise, any name-value pair in one set appears in the result. In the above example, the variable N is unified with noun, and X with the feature set associated with type in the second feature set. Note that order of name-value pairs is not important.

There are at least two ways of implementing feature sets in logic programming. One method involves a global analysis for all names used in an entire grammar and assigning a positional argument to this name, these positional arguments being attached (in the manner of DCGs) to all nonterminals used in the grammar, thus directly mapping the extended unification into Prolog's unification (the details being hidden from the user). Another method makes use of a logic programming data structure in which lists are represented using what is called a tail variable: instead of a list ending with the atom [], it ends in a variable. This allows the variable to be instantiated later, thus making the list longer. During extended unification, this technique permits the addition to a feature set of name-value pairs which do not originally appear in it. This technique was first reported in a paper describing the implementation in Prolog of a Lexical Functional Grammar System [10].

Definite Feature Grammars: Generalized Feature Values and Constraints. In Definite Feature Grammars, introduced in [4], feature value sets may be attached to nonterminals to express lexical, syntactic, semantic or pragmatic knowledge about nonterminals. Generalized unification permits the combination of such knowledge from various nonterminals during parsing or generation. One original aspect of DFGs is that feature values are generalized so that they may be any logic programming (or Prolog) term; furthermore, in specifying feature values, it is possible to write constraints on the values as a conjunction of goals to be solved. Since such constraints on features look as if a definite clause is being defined, the formalism is called Definite Feature Grammars.

DFG rules may, like earlier logic grammar formalisms, be compiled by simple methods into logic programming clauses. We chose to compile to NU Prolog, a sophisticated implementation of logic programming which contains a very powerful and general mechanism for delaying goals in case certain information is not yet available. The

mechanism consists of writing "when" declarations which prevent a goal from executing until some Boolean combination of variables in the head of a clause is sufficiently grounded. We are thus able to initiate the solution of these DFG constraint goals at unification time, and to automatically suspend them until enough information becomes available, and thus get something which may be considered a Constraint Logic Grammar formalism. An advantage to such an approach is that the constraints may be written in a more declarative style, with less attention to control and order as required when using DCGs and other earlier logic grammar formalisms. (Since NU Prolog is in many ways the closest practical approach to the ultimate goal of *Pro*gramming in *Log*ic, there are other advantages to basing a logic grammar formalism on it, including safe negation, extended logical expressions, etc. See [14],[15]

We describe the syntax and semantics of Definite Feature Grammars by way of the classic example which [12] used to introduce the notion of attribute or property grammars to the world. The following formal grammar with start symbol "number" defines a language of bit strings, such as "101.01" and associates with number a value representing the bit string as a decimal number. The value is specified by associating with each bit in the string the value that bit has when the bitstring is considered as the binary positional notation for a number. That number is specified in terms of the power of two associated with that particular place in the string, and that particular place for the integer portion of the value depends on the length of the bitstring to the right of the relevant bit. This is specified using a DFG as follows:

```
bit::{value=0} --> "0".

bit::{scale=S,
    (value=  V :- power(S,V))
    } -->
      "1".

bitstring::{length=0,value=0} --> [].

bitstring::{ (scale=Scale :- S1 is Scale - 1),
        (length=Length :- Length is 1 + L1),
        (value=Value :- Value is VB + V1)
        } -->
  bit::{value=VB,scale=Scale},
  bitstring::{length=L1,value=V1,scale=S1}.

number::{ (value = V :- V is VB + VF)} -->
  bitstring::{length=Length,value=VB, (scale=S :- S is Length - 1)},
  fraction::{value=VF}.

fraction::{value=V} -->
   ".", bitstring::{value=V, (scale = S :- S is -1)}.

fraction::{value=0} --> [].

power(B,C) :- C is 2 ** B.
```

The nonterminals are bit, bitstring, fraction and number; the terminal symbols are "0", "1" and ".". A feature set {...} is associated with a nonterminal by the operator "::". Associated with the nonterminals number, fraction, bitstring and bit is a feature set with feature name "value"; also associated with the nonterminal bitstring are feature names "length" and "scale", and with the nonterminal "bit" is associated the feature name "scale". Some of the specifications of features are written:\

name = Value :- *Goals*

This is understood to mean that the eventual value of "name" is "Value" provided that the *Goals* are satisfied, instantiating possibly, variables in Value. (We call our formalism Definite Feature Grammars because the format of the constrained feature specification looks like a definite clause specifying the "relation" name = Value.) In the example, the goals include various arithmetic constraints used to determine the value of bits, bitstrings, fractions and numbers. *Note that we declare the relations between these quantities but do not specify anything about what order these quantities are computed in.* Because we are implementing this in NU Prolog where arithmetic is delayed until two arguments of an arithmetic operator are grounded, we need not concern ourselves about ordering the computation. This may be compared with a similar example used to introduce the author's earlier Definite Clause Translation Grammars [1] in which because of the Prolog in which it was implemented, one had to be careful to make sure that lengths were known before computing scales in the integer portion of the number. One could, when using an "ordinary" Prolog get around this by creating a structure which is evaluated at the end of parsing: in this example, a large arithmetic expression would be created and then evaluated. The drawback to doing so however, is that if the value is being used to constrain parsing, one has to complete a parse even though it may be subsequently filtered out by a constraint. (It would be possible to make use of the delaying mechanism even in a NU Prolog implementation of DCGs, but a grammar formalism with feature structures is more perspicuous.)

As a result of parsing the string "10", we obtain the following annotated derivation tree:

```
0: number(4) {yield="10",
              value=2.0}
  1: bitstring(3) {length=2,
                   value=2.0,
                   scale=1,
                   yield="10"}
    2: bit(1) {value=2.0,
               scale=1,
               yield="1"}
      3: "1"
    2: bitstring(3) {length=1,
                     value=0,
                     scale=0,
                     yield="0"}
      3: bit(0) {value=0,
                 scale=0,
                 yield="0"}
        4: "0"
      3: bitstring(2) {length=0,
                       value=0,
                       scale=-1,
                       yield=[]}
        4: []
  1: fraction(6) {value=0,
                  yield=[]}
    2: []
```

Numbers in brackets attached to nonterminals, such as number(4) give the compiled rule number. Feature sets follow the nonterminal to which they are attached. This tree structure with attached feature sets containing attribute-value pairs, exactly mirrors the original model of Knuth's attribute grammar derivation tree except in that the values are computed by logic programming rather than with a functional notation. Note the feature name "yield" whose value is the terminal string generated by the nonterminal. See the section on **Metagrammatical programming and DFGs** for an explanation of this feature.

Implementation of Definite Feature Grammars. We outline the translation of a DFG rule into Prolog (assuming NU Prolog). First of all, each nonterminal will be translated into a three place predicate, for example:

```
number(Tree,Input,Output)
```

where the first argument is a derivation tree, and the following two arguments the two components of the difference list representing the string being analysed or generated.

The derivation tree is represented by a three argument function symbol:

```
node(Label, Features, Subtrees)
```

The Label is the name of the nonterminal with an attached index giving the number of the rule (used for debugging purposes). On the left hand side, the value of Features is a logical variable which will be unified with any specified feature set - in a call, the logical variable may already be instantiated to a feature set from the calling nonterminal; in the right hand side the value of Features is the feature set compiled to a list with a tail variable as in [10]. The value of Subtrees is a list of the subtrees of this node, one for each element, including terminals in the right hand side of the grammar rule.

Consider the rule for number from the bitstring example:

```
number::{ (value = V :- V is VB + VF)} -->
  bitstring::{length=Length,value=VB, (scale=S :- S is Length - 1)},
  fraction::{value=VF}.
```

Here is the Prolog clause corresponding to the rule for number:

```
number(node(        number(4),
                    Features,
                    [node(B,
                          [length = Length, value = VB, scale = S|_],
                          Subtrees_bitstring),
                     node(I, [value = VF|_], Subtrees_fraction)
                    ]),
           Input, Output) :-
    difference(Input, Output, String),
    unify([string = String|_], Features),
    V is VB + VF,
    unify(Features, [value = V|_]),
    S is Length - 1,
```

```
bitstring(node(B,
                [length = Length, value = VB, scale = S|_],
                Subtrees_bitstring), Input, T),
fraction(node(I, [value = VF|_], Subtrees_fraction), T, Output).
```

The predicate difference computes the standard list representation of the difference list Input - Output. An internally created feature set {yield=String} is unified with Features (see **Metagrammatical programming and DFGs** for some applications of this feature), the computation of V as the sum of VB and VF is initiated and then {value=V} is unified with Features. Corresponding to the right hand side, the evaluation of S as Length - 1 is initiated, and then the predicates bitstring and fraction are called in succession, bitstring recognizing a string between Input and T, and fraction using up the string from T to Output. This compiled predicate will fail unless, as in NU Prolog, the arithmetic predicates suspend until at least two arguments are grounded. The predicate difference is written so that it will suspend until Input and Output are sufficiently grounded. Note that the user may have to provide information in the form of NU Prolog's "when declarations" to insure proper execution of the compiled code: this information, however, is easy to provide and is the sole contribution that the user has to make as far as control is concerned.

Metagrammatical programming and DFGs. The notion of a metanonterminal *meta* was introduced in [2] in order to provide a simple means of extending the underlying grammatical formalism and to deal with various linguistic phenomena. Since then, there have been some implementations of Prolog which have a built in metanonterminal (*parse* of Quintus and Sicstus), and some other researchers have developed a number of other interesting applications of grammatical metaprogramming, including a grammatical treatment of the analysis of DNA strings. (See [20],[18]). Thus, grammatical metaprogramming is widely recognized as a useful technique and we provide it in our implementation of DFGs. Because DFGs make use of feature sets, and because feature values may be any logic programming term, the applications of metaprogramming turn out to be rather cleaner and easier than metaprogramming in DCTGs. In this section we will provide a few examples of grammatical metaprogramming in the DFG formalism.

Here is a definition which extends the formalism by allowing the specification of a sequence of grammatical symbols:

```
seq::{ list = [] } --> [].

seq::{ kind = X , list = [Head|Tail] } -->
        X::{yield = [Head]},
        seq::{ kind=X, list = Tail }.
```

A sequence of kind X is either empty or it consists of a metaoccurence of an X followed by a sequence of kind X. To represent the metaoccurence of an X with certain feature values specified, we write X::{yield = [Head]} as above. To represent the metaoccurence of an X without any specified feature values, it is enough to write X. During the compilation of DFG rules to logic programming clauses, suitable calls to our system predicate '$meta' (see Appendix) are generated for any metaoccurence of a symbol. To represent the "value" of a sequence, there is a feature called "list" which forms a list of the individual elements of the sequence. The empty sequence contributes the empty list as value, and a nonempty sequence combines a Head from the metaoccurence of X and the Tail from the subsequence. The special feature name "yield" is automatically attached to all nonterminals: its value is the string derived from that nonterminal.

all nonterminals: its value is the string derived from that nonterminal.

A general conjunction rule is given by:

```
conj::{ kind=[Begin,Left,Middle,Right,End],
        operator=Op,
        (value = V :- combine(Op,V1,V2,V) )} -->
        Begin,
        Left::{value=V1},
        Middle,
        Right::{value=V2},
        End.
```

Here a conjunction is specified by the feature names "kind", "operator" and "value". The value of kind is a quintuplet specifying punctuation symbols "Begin", "Middle" and "End" which demarcate the conjoined "Left" and "Right" elements. The operator "Op" specifies how the values of the conjoined elements are to be combined by the goal combine(Op,V1,V2,V) to specify the value V of the conjunction.

Assuming that a grammar has been defined with start symbol sentence, and with a feature "value" whose value is the logical form of the sentence, we can define one kind of conjunction as follows:

```
conj::{kind= [[either], sentence, [or], sentence, []],
        operator=or}
```

Here, Right is the empty string. Another kind of conjunction is defined by:

```
conj::{kind= [[], sentence, [and], sentence, []],
        operator=and}
```

In this case, both the Left and Right punctuation symbols are empty. For a formal application of the conjunction rule, here is a definition of how two numbers from our example given above could be written:

```
conj::{kind= ["(", number, "+", number, ")"],
        operator=sum}
```

In order for these conjunctions to be correctly analysed, the following rules for "combine" must be provided:

```
combine(sum,A,B,C) :- C is A + B.
combine(or,A,B,or(A,B)).
combine(and,A,B,and(A,B)).
```

The top level of a derivation tree for the conjoined sentence "Either John loves Mary or Mary loves a cat" is given by:

```
0: conj(13) {kind=[[either], sentence, [or], sentence, []],
                    operator=or,
                    yield=[either, john, loves, mary, or,
                                mary, loves, a, cat],
            value=or(loves(john, mary),
                    some _UPZI &(cat(_UPZI),
                                loves(mary, _UPZI)))}
        1: leaf(either) {yield=[either]}
        1: sentence(7) {value=loves(john, mary),
                        yield=[john, loves, mary]}
        1: leaf(or) {yield=[or]}
        1: sentence(7) {value=some _UPZI &(cat(_UPZI),
                                loves(mary, _UPZI)),
            yield=[mary, loves, a, cat]}
        1: empty {yield=[]}
```

A number of applications given in the papers of Wilmes and Searles involve constraints on the yield or on a part of the yield. A simple example is the definition of a rule for a palindrome for some grammatical category X:

```
palindrome::{kind=X} -->
        X::{yield = S},
        X::{yield = R :- reverse(R,S)}.
```

A string of repeated input is specified by:

```
repeat ::{kind=X} -->
        X::{yield = S},
        X::{yield = S}.
```

Other formal examples given in these papers can be constructed by more complex constraints on the value of the feature yield. When necessary, new operator definitions could be given as in Wilmes' or Searles' work for special purpose grammar formalisms, but the current extensions of '$meta suffice for their implementation.

Both Wilmes and Searles introduce a notion of assignment of the derived string. For example, Wilmes introduces the notation:

```
item =: vol
```

where *item* represents a metacall of an allowable item (eg, nonterminal, terminal, list, etc.) and *vol* is a variable or a list which gets instantiated to the string derived from *item.* This may be simply expressed in DFG terminology as:

```
Item::{yield=Vol}
```

One interesting relation which can also be defined by means of the metaextensions is the notion of derivability. In our terms this is:

```
Item ==>* String :- '$meta'(Item::{yield=String},Tree,String,[])
```

As Wilmes notes, an application of the notion of derivability may be used to specify what it means for a string to be in the intersection of languages (left here as an exercise for the reader). Perhaps a more interesting application of the notion of derivability, coupled with our metaextensions, is the possibility of giving a metadefinition of derivability in Dahl's Static Discontinuous Grammars (see our concluding remarks).

ID/LP parsing. Ordinary context-free rules specify both the constituents of each syntactic category and the order in which they occur. This is suitable for many situations, particularly for languages with a fixed word order but is not very suitable for languages where there is flexibility in word and/or constituent order. Such languages can be more easily described by "factoring out the immediate dominance and linear precedence relations, and stating them separately." (See [11]. Such separation of immediate dominance (ID) and linear precedence (LP) constraints has an extensive history which is sketched in [7].) For example, the ordinary context-free rules:

```
a --> b c d
b --> a c d
c --> a b d
d --> a b c
```

do not capture the generalization that in the specified language sister constituents always appear on the right hand side in alphabetical order, whereas the following ID/LP rules (where right hand sides are considered to be unordered) do:

```
ID
    a --> {b, c, d}
    b --> {a, c, d}
    c --> {a, b, d}
    d --> {a, b, c}

LP      a < b < c < d                Example 1.
```

ID/LP rules also can be more concise than context-free rules. The following ID/LP rule with empty LP constraints would require 120 separate context-free rules (one for each possible permutation of *a, b, c, d, e*) to specify the same syntactic category *s*:

```
s --> {a, b, c, d, e}                Example 2.
```

Given the convenience of ID/LP rules, how does one parse with them? We present a very simple metarule which extends top-down parsing of logic grammars so that ID/LP rules may be handled.

Metarules for UCFG parsing. We first give metarules which permit the parsing of UCFG rules, i.e., ID/LP rules with no LP component. These simply specify that a syntactic category is given by some permutation of the constituents in the right hand side of a rule. Using such rules we can specify free word or constituent order phenomena (but

not free constituent order phenomena where the constituents may be discontinuous). We use the !/1 symbol to name our metanonterminal.

```
!([]) --> [].
!([X|Xs]) -->
    { select(Z, [X|Xs],Ys) }, meta(Z), !(Ys).

select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :-
    select(X, Ys, Zs).
```

The first rule for !/1 is used to recognize an empty permutation as the empty string. The second rule for !/1 nondeterministically selects *Z* from the non-empty list *[X/Xs]* of grammatical symbols to be permuted leaving *Ys*, parses a *meta(Z)* and then recursively parses with a permutation of the remaining grammatical symbols *Ys*. The predicate *select* nondeterministically selects *X* from the second argument yielding the remaining elements in the third argument.

Here is how Example 2 would be written:

```
s --> !([a,b,c,d,e]).
```

provided that we define

```
a  --> [a].          a --> [].
b  --> [b].          b --> [].
c  --> [c].          c --> [].
d  --> [d].          d --> [].
e  --> [e].          e --> [].
```

Here with slight cheating is an example of free word order treatment of a somewhat contrived Latin sentence meaning "Good girl loves small boy":

```
latin --> !([noun::{case=acc}, adj::case=acc},
    noun::{case=nom}, adj::{case=nom}, verb]).
verb --> [amat].
noun(nom) -->[puella].          noun(acc) --> [puerum].
adj(nom) --> [bona].            adj(acc) --> [parvum].
```

The cheating lies in not being able to treat in this formalism the discontinuous constituents of the nominative and accusative noun phrases *puella... puerum* and *bona... parvum.*

This method of specifying when constituents may be scrambled can be combined with order dependent parts of rules. Highly simplifying, in Japanese, the subject, direct object and indirect objects may be scrambled, but the verb must appear at the end of a sentence. Also, the subject and objects may be elided. The following are legal Japanese sentences:

```
taro ga hon wo hanako ni ataeta.
hon wo hanako ni taro ga ataeta.
Taro gave the book to Hanako.
```

taro ga hanako ni ataeta.
Taro gave (something) to Hanako.

taro ga ataeta.
Taro gave (something to someone).

ataeta.
(Someone) gave (something to someone).

The post positional particles ga, wo and ni mark the subject, direct object and indirect objects, respectively. The following grammar generates all the above sentences and a number of others which are not shown.

```
s --> !([option::{kind=subject}, option::{kind=obj1},
         option::{kind=obj2}]), [ataeta].
subject --> [taro],[ga].
obj1 --> [hon],[wo].
obj2 --> [hanako],[ni].
```

Here option is a metarule which finds an optional occurence of the nonterminal argument to the feature value kind:

```
option::{kind=X} --> X.
option::{kind=_ } --> [].
```

This example was motivated by a discussion of scrambling and ellipsis in Sugimura* .

Metarules for ID/LP parsing. In order to deal with nonempty LP constraints we have to test whether adding the selected element to the list of grammatical symbols which have been used so far in an ID/LP rule does not violate the LP constraints. To simplify the checking of constraints we do some preprocessing. For each symbol x which appears in a linear constraint we can derive a relation which specifies what is forbidden in a parse. Thus for the constraint

$$a < b < c$$

we get

```
forbidden(b,a).
forbidden(c,b).
forbidden(c,a).
```

That is, it is forbidden for a *b* to occur before an *a*, or for a *c* to occur before either a *b* or an *a*. The last definition of *forbidden* is derived by considering the transitive closure of what is directly forbidden by the linear constraint.

---

* R. Sugimura. A Parser for Scrambling and Ellipsis. In particular, the handling of these problems with respect to the SAX system (see [13]).

Further, we specify that a constituent is constrained if it occurs in a forbidden relationship, and that two constituents *A* and *B* are allowed to occur in that order as long as it is not forbidden:

```
constrained(X) :- forbidden(X,_),!.
constrained(X) :- forbidden(_,X).

allowed(A,B)   :- not(forbidden(A,B)).
```

We are then able to define the following rules for parsing with nonempty LP constraints.

```
!(History,[]) --> [].
!(History,[X|Xs]) -->
    { select(Z,[X|Xs],Ys),
     check_selection(Z,History),
     append(History,[Z],NewHistory) },
     meta(Z),
     !(NewHistory,Ys).

check_selection(Sel,SoFar) :-
    not( constrained(Sel)).
check_selection(Sel,SoFar) :-
    constrained(Sel),
    check_constraints(SoFar,Sel).

check_constraints([],Sel).
check_constraints([X|Xs],Sel) :-
    constrained(X),
    allowed(X,Sel),
    check_constraints(Xs,Sel).
check_constraints([X|Xs],Sel) :-
    not(constrained(X)),
    check_constraints(Xs,Sel).
```

In the definition of !/2 the first argument is a history of which constituents have already been parsed and the second argument is a list of the constituents which have yet to be recognized. The first rule for !/2, used to terminate a parse with an ID/LP rule, specifies that an empty list of constituents to be recognized generates the empty string. In the second rule for !/2, *Z* is nondeterministically selected from the set of constituents yet to be parsed and *check_selection* verifies that no constraints are violated. A new history is constructed by appending the old history and *[Z]*, a metacall is made to parse a *Z*, and then !/2 is recursively called with the remaining constituents to be recognized. The predicate *check_selection* succeeds if the selected element is not constrained, otherwise it succeeds if it is constrained and *check_constraints* succeeds. The first argument of *check_constraints* is the history of what has been parsed so far and the second argument is the selected element. The predicate *check_constraints* succeeds: if the history is empty; if the first element *X* of the history is constrained, allowed(X,Sel) succeeds and the remaining constraints can be checked; or, if the first element of the history is not constrained, and the selected element does not violate any constraints in the remaining elements of the history.

The following grammar generates all permutations of a, b, c and d in which a must precede b:

```
s --> !([],[a,b,c,d]).
a-->[a].
b-->[b].
c-->[c].
d-->[d].
a < b.
```

The unit clause forbidden(b,a) is generated from the constraint a < b. The goal s([],[a,b,c,d],Tree,X,[]) will successively generate in X each of the allowed permutations. *Tree* represents the derivation tree for such a permutation.

Here is an example which may be found in [3].

```
s  --> !([],[np,vp]).
vp --> !([],[v,pp::{yield=with},pp::{yield=in}]).
np --> [gustave].
np --> [imogen].
np --> [blackpool].
v --> [lives].
pp::{yield=with} --> !([],[[with],np]).
pp::{yield=in} --> !([],[[in],np]).
forbidden(vp,np).
forbidden(pp(With_or_In),v).
forbidden(N,[X]) :- member(N,[s,vp,np,v,pp(_)]).
```

The definitions of *forbidden* are derived from the following constraints:

```
np < vp
v  < pp(in)
v  < pp(with)
No nonterminal may occur before a terminal symbol in a rule.
```

Concluding remarks. In this paper we have presented a number of developments of logic grammars. The work on ID/LP parsing is taken from [3] in which there are also comparisions to another ID/LP parsing algorithm due to Shieber (described in [7]) which generalizes Earley's parsing algorithm. The main drawback to using top down recursive descent parsing methods is that left recursive grammar rules present a problem. In adapting recursive descent parsing to deal with ID/LP rules, recursion can also present an obstacle to processing. Consider the rules:

```
s --> [].
s --> !([a,s]).
a --> [a].
```

Under certain orderings of these rules, and certain choices of elements in the right hand side of the second rule, Prolog would go into an infinite loop. Even if a first parse is obtained, attempting to show that it is the only parse might result in an infinite loop. The problem of left recursion, however, does not prevent top down parsing methods from

being useful in many practical situations. In a recent paper [6] the method described here is refined, and another method which makes use of Dahl's Static Discontinuity Grammars [8] is described. Using the latter method it is fairly easy to add loop control as a constraint on parsing. It should also be possible to use pure grammatical metaprogramming on DFGs for loop control. Another strategy to control left recursion adopted by [9] in the Constraint Logic Grammar system makes use of the fact that their grammars do not permit erasure and so any rule application must consume input. Thus, limits can be set as to the depths of recursion. It should be possible to adapt their method to a more general class of grammars in which erasure is permitted, but which requires that any recursive cycle of nonterminals consumes at least one input token.

References.

[1] Abramson, H. (1984) Definite Clause Translation Grammars. Proceedings of the IEEE Logic Programming Symposium, Atlantic City, pp. 233-240

[2] Abramson, H., Metarules and an approach to conjunction in Definite Clause Translation Grammars. Proceedings of the Fifth International Conference and Symposium on Logic Programming. Kowalski, R.A. & Bowen, K.A. (editors), MIT Press, pp. 233-248, 1988.

[3] Abramson, H. Metarules for Efficient Top-down ID-LP Parsing in Logic Grammars,Technical Report TR-89-11, University of Bristol, Department of Computer Science, 1989. Presented at the Workshop on Prolog as an Implementation Language for Natural Language Processing, Strängnäs, Sweden, April, 1989.

[4] Definite Feature Grammars for Natural and Formal Languages, Proceedings Third International Conference on Natural Language Processing and Logic Programming, Norrköping, Sweden, January 23-25,1991, pp. 222-238. Final version to be published by North-Holland.

[5] Abramson, H., Dahl, V. Logic Grammars. Springer-Verlag, 1989.

[6] Abramson, H. and Dahl, V., On Top-down ID-LP Parsing With Logic Grammars, submitted for publication.

[7] Barton, G.E., Berwick, R.C. and Ristad, E. Computational Complexity and Natural Language, MIT Press, 1987.

[8] Dahl, V. Discontinuous Grammars. *Computational Intelligence*, vol. 5, no. 4, pp. 161-179, 1989a.

[9] Damas, L. and Varile, N. A Guide to Constraint Logic Grammar, MK2A Document Set, Eurotra - P, 1989.

[10] Eisele, A., Dorre, J. A Lexical Functional Grammar System in Prolog. Proceedings COLING 90, Bonn.

[11] Gazdar, G., Klein, E., Pullam, G., Sag, I. Generalized Phrase Structure Grammar. Basil Blackwell, 1985.

[12] Knuth, D.E., Semantics of Context-Free Languages. Mathematical Systems Theory, vol. 2, no. 2, pp. 569-574, 1968.

[13] Matsumoto, Y., Sugimura, R. A Parsing System Based on Logic Programming. *Proc. International Joint Conference on Artifical Intelligence*, 1987.

[14] Naish, L. Negation and Control in Prolog, Springer-Verlag, 1985.

[15] Naish, L. Negation and Quantifiers in NU-Prolog. Proceedings of the Third International Conference on Logic Programming. MIT Press, 1986.

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

[17] Pereira, F.C.N. (1981) Extraposition Grammars. American Journal of Computational Linguistics, vol. 9, no. 4, pp. 243-255.

[18] Searles, D.B. Investigating the Linguistics of DNA with Definite Clause Grammars. Proceedings of the North American Conference on Logic Programming, vol. 1, pp. 189-208, MIT Press, 1989.

[19] Sharp, R. CAT2 -Implementing a Formalism for Multi-Lingual MT. Proceedings of the 2nd International Conference onn Theoretical & Methodological Issues in Machine Translation of Natural Languages, Pittsburgh, Pa. 1988

[20] Wilmes, T. A Generalized Approach to Metaprogramming in Logic Grammars. To appear in New Generation Computing, 1990.

Appendix: Definitions for the predicate '$meta'.

```
'$meta'((A,B),node('?',Feature,RMetaTree),Input,Output) :-
        makelist((A,B),Metalist),
        compile_FD({yield=S},Feature,_),
        difference(Input,Output,S),
        '$metalist'(Metalist,[],MetaTree,Input,Output),
        reverse(MetaTree,RMetaTree).

'$meta'([],node(empty,Feature,[[]]),Input,Input) :-
        compile_FD({yield=[]},Feature,_).

'$meta'([Terminal],node(leaf(Terminal),Feature,[[]]),Input,Output) :-

        compile_FD({yield=[Word]},Feature,_),
        c(Input,Terminal,Output).

'$meta'((Nonterminal::Feature),Tree,Input,Output) :-
        isNonemptyAtom(Nonterminal),
        compile_FD(Feature,FeatureList,Constraints),
        Tree = node(_,FeatureList,_),
        NewNonterminal =.. [Nonterminal,Tree,Input,Output],
        Constraints,
        NewNonterminal.

'$meta'(Nonterminal,Tree,Input,Output) :-
        isNonemptyAtom(Nonterminal),
        '$meta'((Nonterminal::Feature),Tree,Input,Output).
```

The first clause in the definition of '$meta is an extension which allows something like the dynamic parsing or generation using a rule whose right hand side is (A,B) and which is given a nonterminal name "?". So, if a rule such as the following

```
? --> nt1, nt2, ..., ntk
```

did not exist in the original grammar, but were derived dynamically, it would be metacalled as follows:

```
'$meta'((nt1,nt2,...,ntl))
```

This is equivlent to one of Wilmes additions to the repertory of metaprogramming techniques. In the definition of '$meta, (A,B) is turned into a list, the feature representing

the derived string S { yield = S} is compiled by the predicate compile_FD** , and then '$metalist and reverse are used (see below) to create a tree node with name "?".

The second clause of '$meta' specifies what happens with a metacall to recognize an empty string.

The third clause in the definition of '$meta deals with a metacall of a terminal symbol, creating a branch labeled "leaf" and with the terminal string as the value of the feature name "yield".

The fourth clause in the definition of '$meta handles a metacall of some nonterminal to which is attached a set of features, possibly containing constraints. The feature is compiled into FeatureList (a list with a tail variable) and the constraints are gathered into Constraints. An appropriate tree structure is created, and then it and the necessary components of the difference list are appended and the metacall is set up. Execution of the constraints is initiated, followed by the metacall.

The remaining clause in the definition of '$meta handles a nonterminal which does not have an attached feature set, and reverts to the previous definition of '$meta.

```
'$metalist'([],MetaTree,MetaTree,X,X).

'$metalist'([A|B],Trees,MetaTree,Input,Output) :-
        '$meta'(A,ATree,Input,X),
        '$metalist'(B,[ATree|Trees],MetaTree,X,Output).
```

In addition to the definition of *'$meta,* we also provide '$metalist which permits the specification of the metaoccurrence of everything in a given list and a low level definition of *sequence* which, in contrast to the metagrammatical definition of sequence, produces a flat derivation tree which is perhaps more appropriate.

The predicate '$metalist is straightforward. Each item in the list is metacalled and a list of derivation trees is formed as the result.

```
sequence(A,node(sequence,RTree,_),Input,Output) :-
        '$metaseq'(A,[],Tree,Input,Output),
        reverse(Tree,RTree).

'$metaseq'(A,Trees,MetaTree,Input,Output) :-
        '$meta'(A,ATree,Input,X),
        '$metaseq'(A,[ATree|Trees],MetaTree,X,Output).

'$metaseq'(_,MetaTree,MetaTree,X,X).
```

The predicate sequence calls '$metaseq which if successful returns a flat list of trees but in reverse order. A call of reverse corrects this order problem. The definition of '$metaseq is straightforward.

** *compile_Feature_Description*, originally written by Randy Sharp as part of the CAT2 machine translation system [19], subsequently altered to incorporate Eisele & Dorre's generalized unification implementation. The third argument, not needed here, is used in our DFG compiler to gather all the constraints attached to features.