DEFINITE CLAUSE TRANSLATION GRAMMARS

by

Harvey Abramson

Technical Report 84-3

April 1984

Definite Clause Translation Grammars

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

ABSTRACT

In this paper we introduce Definite Clause Translation Grammars, a new class of logic grammars which generalizes Definite Clause Grammars and which may be thought of as a logical implementation of Attribute Grammars. Definite Clause Translation Grammars permit the specification of the syntax and semantics of a language: the syntax is specified as in Definite Clause Grammars; but the semantics is specified by one or more semantic rules in the form of Horn clauses attached to each node of the parse tree (automatically created during syntactic analysis), and which control traversal(s) of the parse tree and computation of attributes of each node. The semantic rules attached to a node constitute therefore, a local data base for that node. The separation of syntactic and semantic rules is intended to promote modularity, simplicity and clarity of definition, and ease of modification as compared to Definite Clause Grammars, Metamorphosis Grammars, and Restriction Grammars.

1. Introduction

A grammar is a finite way of specifying a language which may consist of an infinite number of "sentences". A grammar is a logic grammar if its rules can be represented as clauses of first order predicate logic, and particularly, as Horn clauses. Such logic grammars can conveniently be implemented by the logic programming language Prolog: grammar rules are translated into Prolog rules which can then be executed for either recognition of sentences of the language specified, or (with some care) for generating sentences of the language specified.

Metamorphosis grammars, the first class of logic grammars were introduced in [Colmerauer,1978] and were shown to be effective for recognition of sentences of a small subset of natural language, and also in the writing of a compiler for a simple programming language. Definite Clause Grammars, a special case of metamorphosis grammars, were introduced in [Pereira&Warren,1980] and applied to "compiling", i.e., translating a subset of natural language into first order logic. Metamorphosis grammars (M-grammars or MG's) and definite clause grammars (DCG's) have been used to describe several languages, namely ASPLE, Prolog, a substantial subset of Algol-68 (all these in [Moss,1979] and [Moss,1981]); the first large scale use of DCG's for a natural language application appeared in [Dahl,1981]; more recently DCG's have been used to define a functional programming language HASL in [Abramson,1983]. See also [Moss,82] for the use of Prolog and logic grammars as tools for language definition. [Warren,1977] is of interest in the application of logic programming to compiler writing: it explicates and extends ideas originally in [Colmerauer,1978], but does not use any grammar notation.

Both M-grammars and DCG's can be used for such complex tasks as the definition of the syntax and semantics of languages by attaching arguments to the non-terminal function symbols. This allows the specification of such context dependent properties as agreement between subject and verb, agreement of the arity of clausal definitions of HASL functions, and the generation of code, be it logical expressions as the meaning of a natural language sentence, or combinators representing the implementation of a function.

As the tasks to which M-grammars and DCG's are applied become more complex and as the size of grammars grows, it is easy for the rules specifying such tasks to become rather unreadable. One argument may be taken up with generating a representation of the parse tree of a derivation, others may be taken up for generating code, others for checking context dependencies, etc. Furthermore, making changes to a grammar during a project may be made troublesome by having to remember which arguments specify what, and by having syntax and semantics too closely mixed. During the course of developing a compiler for a language, the syntactic component of a grammar changes rarely, but the semantic component may be altered more often. Thus, to make changes to the semantics, one has to modify rules which incorporate, perhaps too closely, syntax and semantics. Finally, the extra arguments resemble a coding trick for representing Horn clauses local in scope to a grammatical rule. For example,

```
sentence(Logic,sentence(Noun_phrase,Verb_phrase)) -->
  noun_phrase(X,P1,Logic,Noun_phrase),
  verb_phrase(X,P1,Verb_phrase).
```

is a DCG rule which seems to hide the local clauses:

```
logic(Logic) :- logic(X,P1,Logic) , logic(X,P1).
/* sentence */ /* noun_phrase */ /* verb_phrase */

sentence(Noun_phrase,Verb_phrase) :- Noun_phrase, Verb_phrase.
```

where *logic(Logic)*, *logic(X,P1,Logic)*, *logic(X,P1)* are the logic components of the *sentence*, *noun_phrase*, *verb_phrase*, respectively, and *sentence(Noun_phrase,Verb_phrase)*, *Noun_phrase*, *Verb_phrase* are their respective tree representations.

Recently, several classes of logic grammars have been introduced in order to correct some of these problems, and to promote readability, modularity, simplicity, and ease of modification. Restriction Grammars (RG's) were introduced in [Hirschman&Puder,82]. These grammars consist of context free rules, restrictions on these rules, and an automatically constructed parse tree; the restrictions are usually specified as restrictions on the form of the parse tree. Modifier Structure Grammars (MSG's) were introduced in [Dahl&McCord,83]. These grammars automatically create parse trees too; however, they also attach to nodes of the parse tree simple semantic rules, usually in the form of operators acting on subtrees to automatically build up an analysis structure by specifying how the meaning of a node is determined by the interaction of meanings of subtrees, replacing the need for the restrictions of RG's. Furthermore, MSG's also treat coordination metagrammatically.

The Definite Clause Translation Grammars (or Translation Grammars) which we introduce below have the flavour of each of these. A parse tree is automatically produced to record derivations, but this parse tree does not have the very complicated representation used in Restriction Grammars. It is like the parse tree used in MSG's in structure, and in having nodes labeled both by non-terminals and semantic actions. These semantic actions, however, are specified by zero or more Horn clauses which are interpreted during traversal(s) of the parse tree. These semantic rules constitute a data base local to nodes of the parse tree, providing a high degree of locality of reference. We also provide a notation for parse trees and their traversal which is very simple and which gets away from the complicated sequence of moves which must be used in RG's.

Our Definite Clause Translation Grammars are an extension and generalization of DCG's. Arguments of non-terminals could still be used for semantic attributes as in DCG's, but attributes or properties of nodes of the parse tree can be more clearly specified and computed according to the rules attached to nodes by translation grammar productions. Our translation grammar rules are compiled into Prolog rules with three hidden arguments, two as in DCG's for the lists of symbols being analyzed, and a third to represent the parse tree.

Our translation grammars are modelled on the attribute grammars of [Knuth,68] which until the advent of Prolog and logic programming have been, except under severe restrictions, difficult to implement. (A typical comment about this is: "Implementation of a translation scheme with both inherited and synthesized attributes is not easy." [Aho&Ullman,73] page 778. Inherited attributes are properties of a node of a parse tree which are dependent on the context of the node; synthesized attributes are those dependent on the subtree rooted at the node [Knuth,68]. Given the power of the logical variable, there is, however, little practical need to place too much emphasis on the difference between inherited and synthesized attributes, and so we do not make the distinction in the sequel. Theoretically, the classification of attributes as inherited or synthesized was used by Knuth to determine whether circular definitions of attributes existed in a translation; however, the occurs check would equivalently detect such circularities without having to make the classification explicit.)

In the next section we define Definite Clause Translation Grammars. A section of examples and comparisions follows. A short section describes the compilation of translation grammar rules to Prolog. A concluding section indicates some future work and applications. An Appendix contains the predicates for compiling translation grammar rules into Edinburgh C-Prolog [Pereira,82], based, in fact, on the C-Prolog DCG to Prolog compiler.

2. Translation Grammars, Tree Formation and Traversal

A definite clause translation grammar rule may be of the form:

LeftPart ::= RightPart <:> Attributes ::- Semantics.

The *LeftPart ::= RightPart* portion of the translation rule specifies one step in a derivation of a "sentence" almost exactly like a definite clause grammar rule: the *LeftPart* may consist of a non-terminal, or a non-terminal followed by a list of terminals; the *RightPart* may consist of terminals, non-terminals, and Prolog terms enclosed in braces { and }. In a translation grammar, however, a parse tree is automatically formed to record the derivation of a sentence, so if *NonTerminal* is the non-terminal in the *LeftPart,* there will be a node in the parse tree for each use of this production, each such node labeled by *NonTerminal* and also by the semantic portion of the rule to the right of the <:> symbol. In the *RightPart* furthermore, if *nt* is a non-terminal, it may have attached to it by the ^^ operator a logical variable *NT,* say, which will be instantiated to the subtree of the parse tree corresponding to the sub-derivation of *nt*.

The symbol <:> separates the syntactic and semantic portions of a translation rule.

Attached to each node of a parse tree is a logical clause representing the "properties" of a node in a translation. *Attributes ::- Semantics,* the portion of the translation rule to the right of <:>, may be read declaratively as: the *Attributes* of the node corresponding to this use of the production are specified by the term or conjunction of terms in *Semantics;* or procedurally as: to compute the *Attributes* of this node, compute the goal or conjunction of goals in *Semantics*. If the specification or computation of an attribute of the non-terminal in the *LeftPart* of a translation rule depends on an attribute *attrib(Args)* of a node corresponding to use of a nonterminal *nt* in the *RightPart* of the syntactic portion of a production, and if *NT* specifies the subtree corresponding to the subderivation of *nt,* then the attribute of *NT* is declaratively specified by:

NT^^attrib(Args)

and may be read procedurally as: traverse the subtree *NT* and compute its attribute *attrib(Args)*.

Example.

```
sentence ::= noun_phrase^^N, verb_phrase^^V, { agree(N,V) }
<:>
logic(P) ::- N^^logic(X,P1,P),
        V^^logic(X,P1).
```

Here, a sentence is defined as a *noun_phrase* followed by a *verb_phrase*, with parse trees N and V, respectively. A Prolog predicate *agree* checks numerical agreement between the *noun_phrase* and the *verb_phrase*. The attribute of this production is *logic* which specifies how the *sentence* is represented as a logical expression in terms of the logical expressions of its components. See Translation Grammar 2, next section.

Sometimes a translation production may have a set of semantic rules associated with it in order to specify (compute) different properties of a node. Such a production is written

```
LeftPart ::= RightPart <:> (Semantic , Semantics).
```

where *Semantic* is a Horn clause as specified above, and *Semantics* are other Horn clauses for other attributes. In this case, each *Attributes ::- Semantics* rule specifies declaratively a property of a given node, and procedurally the way to compute a property of the node. The set of semantic rules of a given production constitutes a local data base for each node of the parse tree representing use of that production.

Example.

```
verb ::= [loves]
<:>
agree(singular),
logic(transitive,X,Y,loves(X,Y)),
logic(intransitive,X,loves(X)).
```

This production specifies three attributes of the verb "loves": *agree* which is used to check agreement of noun phrases and verb phrases with respect to number; and two *logic* attributes which specify a logical expression representing this verb used transitively, and intransitively (see Translation Grammar 2 below). Here, the translation rules may be considered a local data base for the syntactic rule

```
verb ::= [loves].
```

3. Examples and comparisions.

3.1. Example 1.

Translation Grammar 1 below is an adaptation to Prolog of one described in [Knuth,68] to illustrate the convenience of having various attributes attached to nodes of a parse tree for specifying semantics. The strings of the language are binary numerals, and the semantics specify the decimal value corresponding to each binary numeral, integer and fraction. The specified semantics is intended to mirror the way we use this notation, that is, as a positional notation wherein each bit represents a value of zero or some power of two. Thus, to each *bit* is assigned a *value* and a *scale* which is written in C-Prolog as, *2^Scale*. The *Scale* of the leading bit of the integral part of the numeral depends, however, on the number of bits in the integral part and this is specified in terms of an attribute *length*. In the semantic rule for *number* the tree B representing the integral *bitstring* is traversed to compute its length from which the scale factor is obtained and used in a subsequent traversal of B to compute its *value*. The scale factor for computing the *value* of the *fraction* does not require a length for its computation.

Note that when there is only a single attribute or attribute list attached to a production, as in the productions for *bit* and *number*, we can use either lists or logical variables alone on the left-hand side of the symbol *::-* or alone as unit clauses.

For comparision, here is a *logic* DCG for essentially the same translation, given in [Moss,1981]. This logic grammar assumes "that the functions are not evaluated" and ignores both "backtracking and left-recursion". Except that the DCG notation may perhaps be more familiar to the reader, we feel that out translation grammar rules are cleaner and more perspicuous. Translation grammar rules also have the advantage that if one wanted to change the semantics it would be possible to do so without having to edit the syntax of the language.

```
N(v)         -> L(v,1,0).
N(v1+v2)       -> L(v1,l1,0); "."; L(v2,l2,-l2).
N(v,l,s)      -> B(v,s).
L(v1+v2,l+1,s) -> L(v1,l1,s+1); B(v2,s).
B(0,s)        -> "0".
B(2*s,s)      -> "1".
```

Here, *N* corresponds to our *number*, *L* to *bitstring*, *B* to *bit*, and * to ^. Variables in this notation begin with a lower case letter, and ; denotes conjunction. It is left to the reader to modify this to a working Prolog DCG.

3.1.1. Translation Grammar 1.

```
bit ::= "0" <:> [0,_].

bit ::= "1" <:> [V,Scale] ::- V is 2 ^ Scale.

bitstring ::= bit^^B, bitstring^^B1
<:>
(length(Length) ::- B1^^length(Length1),
                Length is Length1+1),
(value(Value,ScaleB) ::-
               B^^[VB,ScaleB],
               S1 is ScaleB-1,
               B1^^value(V1,S1),
               Value is VB+V1).

bitstring ::= [] <:> length(0), value(0,_).

number ::= bitstring^^B, fraction^^F
<:>
V ::- B^^length(Length),
     S is Length-1,
     B^^value(VB,S),
     F^^VF,
     V is VB+VF.

fraction ::= ".", bitstring^^B
<:>
V ::- S is - 1,
     B^^value(V,S).

fraction ::= [] <:> 0.

number(Source) :-
   number(T,Source,[]),
   writestring(Source),
    pretty(T),nl, /* prettyprint the tree */
    T^^N,
    write(N),
    nl.
```

3.1.2. Sample Translation.

The following output is the result of satisfying the goal *number("101.01")*.

```
101.01


number
  bitstring
    bit
      [1]
    bitstring
      bit
        [0]
      bitstring
        bit
          [1]
        bitstring
          []
  fraction
    [.]
    bitstring
      bit
        [0]
      bitstring
        bit
          [1]
        bitstring
          []

5.250000
```

3.2. Example 2.

Translation Grammar 2 below is an adaptation of several examples from [Pereira&Warren,1980] and also offers some comparisions with the RG's of [Hirschman&Puder,1982]. The grammar accepts sentences of a small subset of English and translates them to expressions of first order logic. Agreement between *noun_phrase* and *verb_phrase* is specified by an attribute *agree* attached to some translation productions. The translation to logical expressions is specified by one or more attributes *logic* attached to productions.

In [Pereira&Warren,1980], separate DCG's were given to illustrate the checking of a context-sensitive restraint and to illustrate the generation of a translation. We have combined these into one translation grammar to make the following points.

[1] Combining the tasks of the two DCG's into one DCG would have tended to a clutter of arguments attached to non-terminal function symbols. In grammars as small and unambitious as these, the clutter would be manageable; not so in grammars which attempted to deal with a larger subset of English. Translation Grammars clearly separate the various attributes attached to productions and promote readability.

[2] Translation Grammars are true extensions and generalizations of DCG's, so it would have been possible, and in this case more efficient, to use an extra argument attached to nonterminals to check agreement. We have not done so not only to suggest how more complex context-sensitive restraints would be specified as attributes, but also to offer a comparision with RG's. The predicate *agree* which checks agreement in our example is defined simply in terms of traversals of the parse trees *N* and *V* corresponding to *noun_phrase* and *verb_phrase* respectively. RG's do not have a convenient notation for parse trees: so such a restriction would be specified in terms of following a path in the parse tree from one constituent to another. The following predicate, for example, specifies agreement between subject and verb ([Hirschman&Puder,1982], comments removed):

```
subj_verb_agree(Verb,Words) <-
  element(v,Verb,V),
  up(Verb,Predicate),
  coelement(subject,Predicate,Subj),
  element(n,Subj,N),
  (attrb(singular,N,_) -> attrb(singular,V,_);
   attrb(plural,N,_)   -> attrb(plural,V,_);
   true).
```

Here *element (coelement)* scans the children of a node (siblings of a node) for some particular kind of node, and *attrb* checks for some property attached to a terminal or word. The path that is traced is keyed to the grammar in [Hirschman&Puder,1982] which differs somewhat from ours, but basically it is clear that the notation of Translation Grammars is simple and transparent compared to that of RG's.

[3] Translation Grammars provide an easy way of specifying the total meaning of a word by attaching clauses for each meaning or function of a word to a translation grammar production. For example, our rule for the verb "loves" is:

```
verb ::= [loves]
<:>
agree(singular),
logic(transitive,X,Y,loves(X,Y)),
logic(intransitive,X,loves(X)).
```

The first translation rule for *verb_phrase* requires a transitive verb. The predicate *transitive* simply traverses the tree for a verb to check whether there is a clause specifying a transitive logic attribute. There is a similar predicate *intransitive* used by the second form of *verb_phrase*. Consider, however, the RG form of a predicate to determine whether a verb is transitive or intransitive:

```
verb_object(Predicate,Words) <-
% locate v in verb, save in V
  down(Predicate,Verb),down(Verb,V),
% locate n in object and store it in O
  right(Verb,Object),down(Object,O),
% if O is n (noun), then V must have attribute
% 'transitive'
  (test(n,O,O) -> attrb(transitive,V,_);
% otherwise if O is nullobj, V must have
% attribute 'intransitive'.
   test(nullobj,O,O) -> attrb(intransitive,V,_);
   true).
```

Again, Translation Grammar notation is clearer and simpler than RG notation.

[4] Restriction grammars have a representation for the parse tree which has the flavour of a record structure with pointers. In order that all parts of the tree be accessible from any node, there are double links between parent and child nodes, thus violating the occurs check. The representation for parse trees used by Translation Grammars does not violate the occurs check (see next section). It is possible in Translation Grammars to make all information, i.e., the entire tree, available to any node which requires it: the parent tree is simply passed as an argument to an attribute traversal. For example, if in the syntactic part of a translation production there is the following:

```
nt^^NT
```

then in the semantics for the production, *NT* itself could be an argument of a traversal of *NT* to compute some attribute *attrib* of *NT:*

```
NT^^attrib(NT,...)
```

[5] The semantic component of Translation Grammars seems to be more general than that of MSG's. There, it usually consists of some simple globally defined operator (unary or binary) acting on subtrees of a node. Presumably, our notion of attributes as local data bases could be incorporated in MSG's. On the other hand, we treat neither coordination nor left extraposition in Translation Grammars at the moment. However, two more hidden arguments could be used as in [Pereira,1981] so that we could handle left extraposition, and we could presumably use the interpretive parser of MSG's to handle coordination. (See also the next section on implementation for further comments with regard to MSG's).

3.2.1. Translation Grammar 2.

```
sentence ::= noun_phrase^^N, verb_phrase^^V, { agree(N,V) }
<:>
logic(P) ::- N^^logic(X,P1,P),
         V^^logic(X,P1).

noun_phrase ::= determiner^^D, noun^^N, rel_clause^^R
<:>
(agree(Num) ::- N^^agree(Num),
             D^^agree(Num),
             R^^agree(Num)),
(logic(X,P1,P) ::- D^^logic(X,P2,P1,P),
               N^^logic(X,P3),
               R^^logic(X,P3,P2)).

noun_phrase ::= name^^N
<:>
agree(singular),
(logic(X,P,P) ::- N^^logic(X)).

verb_phrase ::= verb^^V, { transitive(V) }, noun_phrase^^N
<:>
(agree(Num) ::- V^^agree(Num), N^^agree(Num1)),
(logic(X,P) ::- V^^logic(transitive,X,Y,P1),
            N^^logic(Y,P1,P)).

verb_phrase ::= verb^^V, { intransitive(V) }
<:>
```

```
(agree(Num) ::- V^^agree(Num)),
(logic(X,P) ::- V^^logic(intransitive,X,P)).

rel_clause ::= [that], verb_phrase^^V
<:>
(agree(Num) ::- V^^agree(Num)),
(logic(X,P1,&(P1,P2)) ::- V^^logic(X,P2)).

rel_clause ::= []
<:>
agree(Num),
logic(X,P,P).

determiner ::= [every]
<:>
agree(singular),
logic(X,P1,P2,all(X,=>(P1,P2))).

determiner ::= [a]
<:>
agree(singular),
logic(X,P1,P2,exists(X,&(P1,P2))).

noun ::= [man]
<:>
agree(singular),
logic(X,man(X)).

noun ::= [woman]
<:>
agree(singular),
logic(X,woman(X)).

name ::= [john]
<:>
logic(john).

name ::= [mary]
<:>
logic(mary).

verb ::= [loves]
<:>
agree(singular),
logic(transitive,X,Y,loves(X,Y)),
logic(intransitive,X,loves(X)).

verb ::= [lives]
<:>
agree(singular),
logic(intransitive,X,lives(X)).

agree(N,V) :-
   N^^agree(Num),
    V^^agree(Num).

transitive(V) :-
    V^^logic(transitive,_,_,_).

intransitive(V) :-
    V^^logic(intransitive,_,_).

sentence(Source) :-
    sentence(T,Source,[]),
    pretty(T),
    T^^logic(Proposition),
    write(Proposition),nl.
```

3.2.2. Sample Translation.

The following output is the result of analyzing and transforming the sentence "Every man that loves loves a woman that loves a man that loves."

```
sentence
  noun_phrase
    determiner
      every
    noun
      man
    rel_clause
      that
      verb_phrase
        verb
          loves
  verb_phrase
    verb
      loves
    noun_phrase
      determiner
        a
      noun
        woman
      rel_clause
        that
        verb_phrase
          verb
            loves
          noun_phrase
            determiner
              a
            noun
              man
            rel_clause
              that
              verb_phrase
                verb
                  loves
```

```
all(_30,=>(&(man(_30),loves(_30)),
  exists(_117,&(&(woman(_117),
    exists(_198,&(&(man(_198),loves(_198)),
      loves(_117,_198)))),loves(_30,_117)))))
```

The variables that are used as existential and universal quantifiers are artifacts of the C-Prolog system.

4. Compilation of Translation Grammars to Prolog.

The compilation of Translation Grammar rules to Prolog is straightforward and is based on the C-Prolog translation of DCG's to Prolog. (This section must be read in conjunction with the Appendix.) The latter is modified so that a third argument is added to each non-terminal function symbol. This argument is instantiated during parsing to a node of the parse tree.

A non-terminal node of the parse tree has the form:

```
node(NT,Nodes,Semantics)
```

where *NT* is the non-terminal term labeling the node, *Nodes* is a list of the nodes corresponding to the right-hand side of the syntactic portion of a translation production with *NT* on the left hand side, and *Semantics* is the set of semantic clauses specifying the attributes in the semantic portion of a translation rule.

Example.
Corresponding to use of the following production in a parse

```
sentence ::= noun_phrase^^N,
         verb_phrase^^V,
         { agree(N,V) }
<:>
logic(P) ::- N^^logic(X,P1,P), V^^logic(X,P1).
```

we would have the following node in the parse tree:

```
node(sentence,[N,V],(logic(P)::-N^^logic(X,P1,P),V^^logic(X,P1)))
```

A terminal node of the parse tree will be a list of terms corresponding to the list of terminal symbols.

Example.
Corresponding to use of the following production in a parse

```
verb ::= [loves]
<:>
agree(singular),logic(transitive,X,Y,loves(X,Y)),
loves(intransitive,X,loves(X)).
```

we would have the following node in the parse tree:

```
node(verb,[[loves]],(agree(singular),logic(transitive,X,Y,loves(X,Y)),
    logic(intransitive,X,loves(X))))
```

The modifications to the C-Prolog DCG compiler are simple. In translating the right hand side of the syntactic portion of a rule, a pair of lists *St* and *StR* of logical variables is maintained. Corresponding to each non-terminal, a logical variable is added to the list *St* to yield a new list

StR. If a logical variable *NT* is associated with a non-terminal *nt* in a production, then it is *NT* which is added to the front of the list *St;* otherwise, an arbitrary logical variable is added on - since the programmer is not interested in traversing this subtree.

In translating the left hand side of a translation production, the third argument to represent a node of the parse tree is formed by copying the left hand side of the syntactic portion of a translation rule "into" *NT* and *Semantics,* respectively, and the list of right hand side nodes is reversed "into" *Nodes,* in the structured term:

```
node(NT,Nodes,Semantics)
```

The representation which we use is essentially that used by MSG's. Although [Dahl&McCord,1983] indicate that two extra arguments could be used for the tree representation in compiling MSG productions into Prolog, we need only one argument for the tree representation.

5. Conclusions and further work.

The main contribution of Translation Grammars is the provision of a convenient, clear, and powerful notation for labeling and traversing derivation trees. This notation is not tied to any particular parsing method, and so, can and should be abstracted from the DCG-type parser used here and applied to other more powerful parsers (e.g., extraposition grammars, MSG's) as a semantic specification notation.

Translation Grammars separate the specification of the syntax and semantics of languages. The computation of the semantic attributes attached to nodes of a derivation tree is generally carried out after syntactic analysis has been completed. It is possible, however, to compute semantic attributes by "tree walking on the fly" (thoroughly mixing metaphors): in the first rule of Translation Grammar 2, for example, the predicate *agree(N,V)* traverses the subtrees for *noun_phrase* and *verb_phrase* in order to enforce the constraint (or restraint, or restriction) of agreement between subject and verb. In this fashion, Translation Grammars can be used like Restriction Grammars, but without the difficulties (pointed out in section 3.2) associated with RG's, to reject (sub)trees which do not satisfy semantic restraints. In Translation Grammars, furthermore, nonterminal symbols may have arguments attached as in MG's and DCG's, but we tend not to favour this style of semantic specification and have not shown any examples using it. There may be situations, however, when judicious use of this technique may be useful.

One aspect of Translation Grammar notation has not been fully exploited. Our Translation Grammar 2 above differs slightly from the DCG's of [Pereira&Warren,1980] in that we do not have a separate production for the transitive verb "loves" and another for the intransitive verb "loves"; rather, we have a single production for the verb "loves" and its different meanings are attached as distinct semantic clauses. The word "loves" could also be used as a plural noun as in "Don Juan's loves". Rather than introduce a production for *noun* which would give this meaning for the word, we can have a single non-terminal for words whose meaning is looked up in a dictionary:

```
word ::= Word, { lookup(Word,Dictionary,Meaning) }
<:>
Meaning.
```

The *Dictionary* would either be globally defined, or perhaps passed as an argument to the parser, and would localize the meanings of a word under a single heading, much as physical dictionaries do. Categories such as *verb* could be defined for example as:

```
verb ::= word^^W, { is_verb(W) }
<:>
logic(P) ::- W^^logic(P).
```

with *is_verb* obviously defined.

In another application we are replacing the DCG parsing and translation of HASL by a Translation Grammar. A functional language such as HASL can be implemented in several ways, not only by [Turner,1979]'s technique which we have used, but also by an SECD machine, or perhaps even by a direct evaluation of a parse tree. A Translation Grammar would provide a syntactic specification of HASL which would remain stable, and the possibility of easily replacing the semantics to test and compare each of these implementation methods.

We have provided in the definition of our traversal predicate an interpreter for executing a local data base of Horn clauses. It would be desireable to see if C-Prolog's built-in indexed data bases could be exploited to more efficiently execute the local clauses. Translation Grammars may also provide a means of specifying how local data bases interact.

6. Acknowledgements.

I would like to thank Veronica Dahl for reading an earlier version of this paper and for her useful suggestions. This work was supported by the National Science and Engineering Research Council of Canada. I must also thank the UBC Laboratory for Computational Vision for time on its VAX running Berkeley Unix: modern and adequate computing facilities are not currently made available to the computer science department by UBC.

7. References.

[Abramson,1983]

Abramson, H., *A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions,* Proceedings Logic Programming Workshop '83, 26 June - 1 July 1983, Praia da Falesia, Algarve, Portugal; also to appear in New Generation Computing.

[Aho&Ullman,1973]

Aho, A.V. & Ullman, J.D., *The Theory of Parsing, Translation, and Compiling,* 2 volumes, Prentice-Hall, 1973.

[Colmerauer,1978]

Colmerauer, A., *Metamorphosis Grammars,* in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[Dahl,1981]

Dahl, V. *Translating Spanish into logic through logic,* American Journal of Computational Linguistics, vol. 13, pp. 149-164, 1981.

[Dahl&McCord,83]

Dahl, V. & McCord, M. *Treating Co-ordination in Logic Grammars,* to appear in American Journal of Computational Linguistics.

[Hirschman&Puder,1982]

Hirschman, L. & Puder, K., *Restriction Grammars in Prolog.* Proceedings of the First International Logic Programming Conference, Marseille, 1982, pp. 85-90.

[Knuth,1968]

Knuth, D.E., *Semantics of Context-Free Languages,* Mathematical Systems Theory, vol. 2, no. 2, 1968, pp. 127-145.

[Moss,1979]

Moss, C.D.S., *A Formal Description of ASPLE Using Predicate Logic,* DOC 80/18, Imperial College, London.

[Moss,1981]

Moss, C.D.S., *The Formal Description of Programming Languages using Predicate Logic,* Ph.D. Thesis, Imperial College, 1981.

[Moss,1982]

Moss, C.D.S., *How to Define a Language Using Prolog,* Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Tittsburgh, Pennsylvania, pp. 67-73, 1982.

[Pereira,1981]

Pereira, F.C.N., *Extraposition Grammars,* American Journal of Computational Linguistics, vol. 7 no. 4, 1981, pp. 243-255.

[Pereira,1982]

Pereira, F.C.N. (editor), *C-Prolog User's Manual,* University of Edinburgh, Department of Architecture, 1982.

[Pereira&Warren,1980]

Pereira, F.C.N. & Warren, D.H.D, *Definite Clause Grammars for Language Analysis,* Artificial Intelligence, vol. 13, pp. 231-278, 1980.

[Turner,1979]

Turner, D.A., *A new implementation technique for applicative languages,* Software - Practice and Experience, vol. 9, pp. 31-49.

[Warren,1977]

Warren, David H.D., *Logic programming and compiler writing,* DAI Research Report 44, University of Edinburgh, 1977.

Appendix: Compilation to Prolog.

```
/* compilation of definite clause translation grammar rules */

:- op(650,yfx,^^).
:- op(1150,xfx,::=).
:- op(1175,xfx,<:>).
:- op(1150,xfx,::-).

translate_rule((LP::=[]<:>Sem),H) :- !,
  t_lp(LP,[],S,S,Sem,H).

translate_rule((LP::=[]),H) :- !, t_lp(LP,[],S,S,Args,H).

translate_rule((LP::=RP<:>Sem),(H:-B)):-
  t_rp(RP,[],StL,S,SR,B1),
  reverse(StL,RStL),
  t_lp(LP,RStL,S,SR,Sem,H),
  tidy(B1,B).

translate_rule((LP::=RP),(H:-B)):-
  t_rp(RP,[],StL,S,SR,B1),
  reverse(StL,RStL),
  t_lp(LP,RStL,S,SR,Args,H),
  tidy(B1,B).

t_lp((LP,List),StL,S,SR,Sem,H) :-
  append(List,SR,List2),
  add_extra_args([node(LP,StL,Sem),S,List2],LP,H).

t_lp(LP,StL,S,SR,Sem,H) :-
  add_extra_args([node(LP,StL,Sem),S,SR],LP,H).

t_rp(!,St,St,S,S,!) :- !.

t_rp([],St,[[]|St],S,S1,S=S1) :- !.
t_rp([X],St,[[NX]|St],S,SR,c(S,X,SR)) :-
  integer(X), X < 256, !, name(NX,[X]).

t_rp([X],St,[X|St],S,SR,c(S,X,SR)) :- !.

t_rp([X|R],St,[[NX|NR]|St],S,SR,(c(S,X,SR1),RB)) :-
  integer(X), X < 256, !, name(NX,[X]),
  t_rp(R,St,[NR|St],SR1,SR,RB).

t_rp([X|R],St,[[X|R]|St],S,SR,(c(S,X,SR1),RB)) :- !,
  t_rp(R,St,[R|St],SR1,SR,RB).

t_rp({T},St,St,S,S,T) :- !.

t_rp((T,R),St,StR,S,SR,(Tt,Rt)) :- !,
  t_rp(T,St,St1,S,SR1,Tt),
  t_rp(R,St1,StR,SR1,SR,Rt).

t_rp(T^^N,St,[N|St],S,SR,Tt) :- add_extra_args([N,S,SR],T,Tt).

t_rp(T,St,[St1|St],S,SR,Tt) :- add_extra_args([St1,S,SR],T,Tt).

add_extra_args(L,T,T1) :-
  T=..Tl,
  append(Tl,L,Tl1),
  T1=..Tl1.

append([],L,L) :- !.
append([X|R],L,[X|R1]) :- append(R,L,R1).

reverse(X,RX) :- rev1(X,[],RX).

rev1([],R,R) :- !.
rev1([X|Y],Z,R) :- rev1(Y,[X|Z],R).

tidy(((P1,P2),P3),Q) :-
  tidy((P1,(P2,P3)),Q).

tidy((P1,P2),(Q1,Q2)) :- !,
  tidy(P1,Q1),
  tidy(P2,Q2).

tidy(A,A) :- !.

c([X|S],X,S).

node(NT,Nodes,((Args::-Traverse),Rules))^^Args :-
  !, Traverse.

node(NT,Nodes,(Args,Rules))^^Args :- !.

node(NT,Nodes,(_,Rules))^^Args :-
  node(NT,Nodes,Rules)^^Args.

node(NT,Nodes,(Args::-Traverse))^^Args :- Traverse.

node(NT,Nodes,Args)^^Args.

:- asserta(( expand_term(T,E) :- translate_rule(T,E) , ! )).
```