Automatic Synthesis of Semantics for Context-free Grammars

Juergen Haas
Bharat Jayaraman

Department of Computer Science
State University of New York at Buffalo
226 Bell Hall
Buffalo, NY 14260

E-Mail: {bharat,haas}@cs.buffalo.edu

June 1991

Abstract

We are investigating the mechanical transformation of an unambiguous context-free grammar (CFG) into a definite-clause grammar (DCG) using a finite set of examples, each of which is a pair $\langle s,\ m\rangle$, where s is a sentence belonging to the language defined by the CFG and m is a semantic representation (meaning) of s. The resulting DCG would be such that it can be executed (by the interpreter of a logic programming language) to compute the semantics for every sentence of the original DCG. Three important assumptions underlie our approach: (i) the semantic representation language is the *simply typed λ-calculus*; (ii) the semantic representation of a sentence can be obtained from the semantic representations of its parts (*compositionality*); and (iii) the structure of the semantic representation determines its meaning (*intensionality*). The basic technique involves an enumeration of parse trees for sentences of increasing size; and, for each parse tree, a set of equations over (typed) function variables that represent the meanings of the constituent subtrees is formulated and solved by means of a higher-order unification procedure. The solutions for these function variables serve to augment the original grammar in order to derive the final DCG. A technique called *partial execution* is used to convert, where possible, the generated higher-order DCG into a first-order DCG, to facilitate efficient bidirectional execution. In the appendix, we provide detailed illustration of the use of such a system for storing and retrieving information contained in natural language sentences. Based on our experimentation, we conclude that an improved version of this system will facilitate rapid prototyping of natural language front-ends for various applications.

1. From Context-Free to Definite-Clause Grammars

1.1 Motivations

The goal of this research is to develop a system that will take as input a *context-free grammar* (CFG) and a finite set of pairs $\langle s,\ m\rangle$, where s is a sentence belonging to the language defined by the context-free grammar and m is the semantic interpretation (meaning) of s, and will produce as output a *definite clause grammar* (DCG) [1] (Pereira and Warren 1980) that will compute the semantics of every sentence of the input grammar. We will clarify in the next subsection the precise sense in which we can solve our stated problem and the research issues it raises, but first we discuss the significance of this problem: (1) Why is it desirable to automatically augment CFGs with semantic interpretation rules? (2) What are the applications of such a system? (3) Are the basic techniques of potential use to other problem areas? We address each of these questions in the remainder of this subsection.

Automatic Augmentation of CFGs: The reason that it is not easy to *manually* augment a CFG with semantic constructors is that the task of building a correct and efficient parsing and generation grammar requires a fair amount of search, with the result that the process is tedious and error prone. Besides, the resulting augmented grammar is not so easy to manually modify, a common operation for the applications we envisage. Even for grammars as small as the one given in the appendix, it is not obvious what the arguments of the grammar rules should be. It is therefore desirable to have a systematic approach for constructing these arguments.

Applications: The system discussed in this paper would facilitate *rapid prototyping* of natural language interfaces for database systems or customizing such interfaces for specific applications (Velardi 1989, Wallace 1984), since the interface could be obtained merely by defining typical input sentences that the system should parse together with their semantic representations. Both the conversion of the natural language query into this representation and the conversion from this representation back into natural language would be handled by the generated interface—the latter operation would be achieved by applying the definite-clause grammar in the reverse direction to the semantic representations. Reversible execution of DCGs is possible because they are essentially (relational) logic programs; it is more feasible to achieve reversible execution for DCGs than for arbitrary logic programs.

Potential Uses: We believe that the techniques needed to develop our proposed system could lead to interesting insights in other areas, although they are not further discussed in this paper: (1) It may provide a new approach to machine learning and automatic programming by combining higher-order deduction with the concept of learning from examples. (2) In combination with a system that learns grammars from examples the proposed research may provide insights into how intelligent agents may efficiently acquire language from examples.

[1] A DCG is a essentially a CFG wherein each nonterminal symbol is enhanced with parameters denoting semantic information. A DCG can be directly converted into a logic program which can be executed to perform parsing or generation. Section 3 provides more details of DCGs.

1.2 Basic Approach

It should be clear that an *arbitrary* transformation (i.e., arbitrary infinite mapping) cannot be inferred from finitely many examples, and hence it is necessary to impose additional constraints on our problem. We make the following three assumptions in order to facilitate the mechanical augmentation of a CFG with semantic interpretation rules: (i) the semantic representation language is the *simply typed* λ*-calculus*; (ii) the semantic representation of a sentence can be obtained from the semantic representations of its parts (*compositionality*); (iii) the structure of the semantic representation determines its meaning (*intensionality*).

These assumptions are not unusual since they have been adopted, for example, by R. Montague in his intensional logic $\mathcal{I}_{\mathcal{L}}$ for a "proper treatment" of quantification of English (Dowty et al. 1981). As it turns out, the search space of allowable solutions is drastically reduced through the use of the simply typed calculus. The compositionality principle effectively means that the grammatical structure constrains the allowable semantics, and the intensionality principle allows us to generalize from examples by generalizing the structures of the semantic representations. The choice of the simply typed calculus allows us to formulate this generalization problem as a unification problem over simply typed terms, for which there are well-developed procedures (Huet 1975, Jensen et al. 1976)—although efficient implementation techniques for them are still under development (Nadathur et al. 1989). Under these constraints, we believe the problem is recursively enumerable in that, if there exists a DCG satisfying the finitely many examples, our proposed system can systematically find it; if there is no solution, the system may sometimes be nonterminating.

Our scheme may be summarized as follows: Given a CFG rule below,

$a \rightarrow b\ c,$

in which lowercase letters are nonterminals, the compositionality principle allows us to enhance the rule as follows:

$a((F\ X\ Y)) \rightarrow b(X),\ c(Y),$

where uppercase letters are variables. That is, if variables X and Y represent the meanings of the nonterminals b and c, respectively, then the meaning of nonterminal a is obtained by applying some function F to X and Y. The function variable F is some λ-abstraction term in the simply typed λ-calculus, and must be determined by the system based upon the finite set of input examples. The semantic representations of the terminal symbols of the CFG are the other unknowns to be determined by the system based upon the input examples.

Briefly, our technique is to enumerate parse trees in some increasing order, query the user for the semantic representation for each of the generated sentences, formulate a set of equations over the function variables that represent the meanings of the constituent subtrees, and solve these equations using the procedure for higher-order unification of typed λ-terms described in Huet (1975). The solutions for these function variables serve to augment the original CFG in order to derive the final DCG. The generation of parse trees terminates when all CFG rules have been enhanced with semantic information. Several research issues arise in this work: (i) The design of efficient implementation techniques for higher-order unification. Even though there are complete enumeration procedures for unifiers, a naive exploration of unifiers will lead to unacceptably poor performance for realistic applications. (ii) The conversion of the constructed higher-order DCG into a first-order DCG where possible. A first-order

DCG is more amenable to reversible execution than a higher-order DCG. We are exploring a technique called *partial execution* which replaces λ-abstractions by first-order terms in those cases where β-reduction amounts to a simple substitution. (iii) The proof of correctness of the transformation when the DCG defines an infinite language. We also plan to investigate for realistic examples the potential sources of nontermination in the transformation process. An experimental system is being developed, and it is shown that such a system facilitates rapid prototyping of natural language front-ends for various applications.

The remainder of this report is organized as follows: section 2 surveys related research; section 3 reviews definite-clause grammars; section 4 describes the basic techniques underlying the system; section 5 discusses partial execution and reversibility; and section 6 presents conclusions and areas of further work. The appendices contain sample runs from the preliminary system that we have implemented.

2. Related Research

We are not aware of any published research that meets our stated objective, but there are several related research areas that bear closely on the techniques needed in our work. We briefly survey these research areas and mention their relationship to our objective.

Natural Language Learning

Lehnert (1987) discusses a system that uses limited syntactic knowledge expressed in a chart parser and relevant conceptual case frame representations as the basis to learn semantic representations of natural language sentences from examples. The system is illustrated by using an "approximation" of conceptual dependency as sample representation (Schank 1975). In order to determine the associations between the words of the sentence and the fragments of the semantic representation, Lehnert's system uses *lexical matches*; i.e., the word has to appear explicitly in the representation, otherwise the association has to be provided by the user in the form of a *conceptual definition*. The semantic representations that can be learned by Lehnert's system are also restricted by the fact that they must be non-recursive case-frame representations.

Other research projects in the area of natural language learning, e.g., (Anderson 1981) and (Selfridge 1981), combine the acquisition of syntactic and semantic knowledge. A major objective of those projects is to explain the characteristics of human language learning. Anderson (1977) discusses a system that infers augmented transition networks given pairs of sentences and structures representing their meanings. The type of meaning representation used by Anderson's system is a propositional semantic network. The augmented transition network inferred by the system can be used for both converting sentences into their semantic representations and vice versa; however, two separate interpreters are required for these two modes of operation, whereas in the case of definite clause grammars only one is needed due to the reversibility property. Overall, Anderson takes a very heuristic approach to language learning, in contrast to the systematic techniques discussed in this proposal.

Machine Learning

Explanation-based learning (EBL) has been investigated mainly in the area of theorem proving, although the same mechanism underlies much of the work in other fields such as skill acquisition and automatic programming. The initial proof that is used to guide the generalization in EBL corresponds to "semitraces" in the area of programming by examples (Smith 1984), and the general proof in EBL corresponds to the synthesized program. In our work, the analysis of a particular sentence and its representation corresponds to the initial proof and is used to guide the augmentation of the grammar.

Determining the association between terminal symbols (words) and their semantic representations using anti-unification can be considered a generalization of learning concepts from examples. By successively considering example sentences whose sentences differ in only one word from the sentence of the "main" example sentence, the system makes use of the powerful concept of near misses (Winston 1975) in its generalization process. Anti-unification is essentially the dual of *most general unification*, which was developed in the context of theorem proving (Robinson 1965). Instead of using an expressive description language to formulate possible generalizations, as in the version space approach (Mitchell 1978), anti-unification generalizes only by turning constants or terms into variables, thus facilitating efficient implementation.

Program Synthesis by Examples

A definite clause grammar can be viewed as a program that takes as input a sentence and computes its semantic representation. Therefore the augmentation process discussed in this proposal can be considered a type of automatic program synthesis from examples (Summers 1977, Bauer 1979, Kodratoff 1979, Biermann et al. 1984). Programming by examples, on the other hand, is a special case of inductive inference, since the synthesis of a program generally involves the inference of an extended pattern of program behavior from the patterns discovered in the examples—computability theory has shown that it is possible to infer large useful classes of programs simply from examples of input/output behavior (Gold 1967, Blum 1975, Barzdin 1977). A common way to deal with the search problem in automatic program synthesis from examples is to use *program schemas* to constrain the way in which the control structures and data operators of the chosen programming language are used (Smith 1984). In the case of the project discussed in this report, the parsing grammar provided to the system can be considered such a program schema. Research in program synthesis from examples has not covered the kind of input/output pairs the proposed project is concerned with.

3. Definite Clause Grammars

To keep this report self-contained, we provide a brief introduction to definite-clause grammars. As noted earlier, definite clause grammars extend context-free grammars with semantic information. From a computational standpoint, they can be used as parsers as well as generators of a language. They differ from context-free grammars in three important respects: (1) grammar nonterminal symbols may include arguments; (2) rules may include predicates/conditions in their bodies; and (3) invocation of grammar rules requires unification of these arguments. Definite clause grammars are actually a particular form of *logic grammars*, other types of which are *metamorphosis grammars* and *extraposition grammars* (Abramson and Dahl 1989).

The following is a definite clause grammar for a trivial subset of English, to illustrate parse tree construction and simple check for number agreement.

```
sentence(sent(N,V)) --> nounphrase(N, Num), verbphrase(V, Num).

nounphrase(np(D,N), Num) --> determiner(D, Num), nounphrase2(N, Num).
nounphrase(np(N), Num) --> nounphrase2(N, Num).

nounphrase2(np2(A,N), Num) --> adjective(A), nounphrase2(N, Num).
nounphrase2(np2(N), Num) --> noun(N, Num).

verbphrase(vp(V), Num) --> verb(V, Num).

determiner(det(the), Num) --> [the].
determiner(det(a), singular) --> [a].

noun(noun(computer), singular) --> [computer].
noun(noun(computers), plural) --> [computers].

adjective(adj(super)) --> [super].
adjective(adj(mini)) --> [mini].
adjective(adj(micro)) --> [micro].

verb(verb(runs), singular) --> [runs].
verb(verb(run), plural) --> [run].
```

To execute the above DCG as a relational program, a typical query might be `sentence(P,[a, micro,computer,runs],[ ])`, which would return in the variable `P` the parse tree for the sentence `[a,micro,computer,runs]`. That is, two extra arguments in addition to those in the rule `sentence` should be supplied, to denote the input list and remaining list (after the parse) respectively. The reversibility property of a DCG is illustrated by the query `sentence(sent(np(np2( super,computers)), vp(run)), S, [ ])`, which returns the sentence `[super,computers,run]` from the given parse tree.

Compositionality and Higher-order DCGs

The DCG shown in the above example is first-order, because the terms that constitute the arguments to each nonterminal come from a first-order universe. However, the *compositional* approach to the semantics of context-free grammars requires the use of higher-order DCGs. In this approach (due to Montague) each phrase is assigned a *logical form*, i.e., a logical expression that has the same truth conditions as the phrase, and the logical forms of the phrases constituting a sentence are systematically composed to construct the logical form for the entire sentence. Montague used a higher-order logic based on the typed λ-calculus as the language for logical forms: the logical form for each phrase is denoted by a typed λ-term, and

β-reduction is used to compose these logical forms. The use of the typed λ-calculus as the universe of terms makes the resulting DCG higher-order (because variables now may stand for functions).

Automatic Synthesis of Semantics

In order to partially automate the synthesis of semantics, various formalisms have been developed within the logic grammar framework, e.g., *modifier structure grammars*, *restriction grammars*, *discontinuous grammars*, and *puzzle grammars* (Abramson and Dahl 1989). These formalisms provide the user with means for specifying guidelines which the system will consult in order to construct the final representation; i.e., the user specifies the desired type of semantic representation in some high level language and the system then translates these specifications into constructors which can be incorporated into the executable grammar rules. The only types of semantic representations that could be completely automated with these approaches are parse trees, since their representations follow exactly the history of rule application. Our work is an attempt to go beyond simple parse tree construction.

4. Automatic Synthesis of Semantics

Section 4.1 briefly describes the general technique and illustrates it by an example; and section 4.2 presents the details of the needed higher-order unification procedure.

4.1 General Technique

We assume, without loss of generality, that all grammar rules have either zero, one, or two non-terminal symbols on the right-hand side, since any CFG can be converted into this form.

4.1.1 Basic Procedure

(0) Input: A CFG where the rules have zero, one, or two symbols on the r.h.s. Add an argument position to each grammar symbol in the following way (if grammar symbols already have arguments, it is simply added as final argument):
if $a \to b\ c$ is a grammar rule, enhance it to $a((F\ A\ B)) \to b(A),\ c(B)$;
if $a \to b$ is a grammar rule, enhance it to $a((F\ A)) \to b(A)$; and
if $a \to [t]$ is a grammar rule, where t is a terminal symbol, enhance it to $a(F) \to [t]$.

(1) Let E be a set of higher-order equations, initially empty.

1a Use the grammar to generate a sentence s_0 involving a minimal number of rules with at least one rule that has not been used before (repeat words from previous sentences if possible to avoid repetitive computations).

1b Generate variations s_i of sentence s_0 in the following way: replace each word w of s_0 by another one of the same syntactic category if w has not occurred already in a previous sentence (under that syntactic category).

1c Query the user for the semantic representation m_0 for s_0 and m_i for each variation s_i.

(2) Execute s_0 on the enhanced grammar. The result n_0 returned in the argument position of the top most rule is a nested application of F_i's. Set up a higher-order equation e_0 of the form $n_0 = m_0$, where m_0 is the semantic representation provided by the user. Similarly set up a higher-order equation e_i for each variation s_i, and add all equations e_i to the set E.

(3) If not all grammar rules have been used yet, go to (1a).

(4) Use the higher-order unification procedure to solve the system of equations given by E for the F_i's and substitute for the F_i's in the grammar rules correspondingly.

4.1.2 Example

We illustrate the above procedure for a very simple CFG, given by the following rules:

```
(r1)  s  --> np, vp.                    (r4)  pn --> [mike].
(r2)  np --> pn.                        (r5)  pn --> [mary].
(r3)  vp --> tv, np.                    (r6)  pn --> [john].
                                        (r7)  tv --> [saw].
                                        (r8)  tv --> [visited].
```

Step (0): These rules are augmented with additional arguments with variables, as described above:

```
(r1)  s((F1 A B))  --> np(A), vp(B).        (r4)  pn(F4) --> [mike].
(r2)  np((F2 A))   --> pn(A).               (r5)  pn(F5) --> [mary].
(r3)  vp((F3 A B)) --> tv(A), np(B).        (r6)  pn(F6) --> [john].
                                            (r7)  tv(F7) --> [saw].
                                            (r8)  tv(F8) --> [visited].
```

Step (1): Generate the sentence `[mike saw mary]` and its variations `[john saw mary]`, `[mike visited mary]`, and `[mike saw john]`, and ask the user for the corresponding semantic representation, which would be (say) `(saw mike mary)`, etc. (Although a type must be specified for each term, often it is possible to work without explicitly given types.)

Step (2): Executing the above DCG on the sentence `[mike saw mary]`, the argument of the starting symbol `s` is instantiated to `(F1 (F2 F4) (F3 F7 (F2 F5)))`. Therefore, we obtain the equation:

```
(e0)   (F1 (F2 F4) (F3 F7 (F2 F5))) = (saw mike mary)
```

Similarly, we get equations for each variation of that sentence (assuming the semantic representations are as given on the rhs of each equation):

```
(e1)   (F1 (F2 F6) (F3 F7 (F2 F5))) = (saw john mary)
(e2)   (F1 (F2 F4) (F3 F8 (F2 F5))) = (visited mike mary)
(e3)   (F1 (F2 F4) (F3 F7 (F2 F6))) = (saw mike john)
```

Step (3): Since all grammar rules have been used by these four training instances we can proceed to step (4).

Step (4): The equations `e0-e3` provide enough constraints to bind the variables F_i such that the resulting DCG shows the desired behavior. We demonstrate in the next section how the higher-order unification procedure would find the bindings for these function variables; for the present, we simply show the substitutions that would be found:

```
F1 = X\Y\(X Y)                     F4 = mike
F2 = X\Y\(Y X)                     F5 = mary
F3 = X\Y\Z\(Y (X Z))               F6 = john
                                   F7 = X\Y\(saw X Y)
                                   F8 = X\Y\(visited X Y)
```

Therefore the resulting (higher-order) DCG is:

```
(r1) s((X\Y\(X Y) A B)) --> np(A), vp(B).        (r4) pn(mike) --> [mike].
(r2) np((X\Y\(Y X) A)) --> pn(A).                (r5) pn(mary) --> [mary].
(r3) vp((X\Y\Z\(Y (X Z)) A B)) --> tv(A), np(B). (r6) pn(john) --> [john].
                                                 (r7) tv(X\Y\(saw X Y)) --> [saw].
                                                 (r8) tv(X\Y\(visited X Y)) --> [visited].
```

4.2 Higher-Order Unification

The terms of the *simply typed* λ*-calculus* are obtained from countable sets of variables and constants having the following types: (i) a constant or a variable of type α is a term of type α, (ii) if v is a variable of type α and b is a term of type β, then $(\lambda v.b)$ is a term of type $(\alpha \rightarrow \beta)$ and is referred to as an *abstraction*, and (iii) if f and x are terms of type $(\alpha \rightarrow \beta)$ and α respectively, then $(f\ x)$ is a term of type β and is referred to as an *application*. We assume the following two elementary types: i (for individuals) and o (for booleans).

4.2.1 Substitution Rules

Suppose that we are given a finite set of pairs of terms of the same type to be unified:

$\{\langle u_1, v_1\rangle, \ldots, \langle u_n, v_n\rangle\}$.

The higher-order unification problem involves finding a substitution σ such that $u_i\ \sigma$ is λ-convertible to $v_i\ \sigma$, where the rules of λ-conversion are as follows: (1) $\lambda V.E \Leftrightarrow \lambda V'.E[V'/V]$ (α-conversion), (2) $(\lambda V.E_1)\ E_2 \Leftrightarrow E_1[E_2/V]$ (β-conversion), (3) $\lambda V.(E\ V) \Leftrightarrow E$, if V has no free occurrence in E (η-conversion). (The notation $E[E'/V]$ is used to mean the result of substituting E' for V in E.) We assume each term is represented in *head-normal* form, as:

$\lambda x_1.\ldots,\lambda x_n.(A\ t_1 \ldots t_m)$,

where A is a constant or variable of type $\alpha_1 \rightarrow \ldots \rightarrow \alpha_m \rightarrow \beta$. A is called the *head* of the term, and the term is said to be *rigid* if A is a constant or is an element of $\{x_1, \ldots, x_n\}$, and *flexible* otherwise. Given two rigid terms

$\lambda x_1.\ldots.\lambda x_n.(F_1\ s_1 \ldots s_i)$ and $\lambda x_1.\ldots.\lambda x_n.(F_2\ r_1 \ldots r_i)$

of the same type, they are unifiable only if F_1 and F_2 are identical, and they can be reduced to:

$$\{\langle \lambda x_1.\ldots\lambda x_n.s_1, \lambda x_1.\ldots.\lambda x_n.r_1\rangle, \ldots, \langle \lambda x_1.\ldots.\lambda x_n.s_i, \lambda x_1.\ldots.\lambda x_n.r_i\rangle\}.$$

Huet observed that either such a set of pairs has no unifier or it can be reduced to another set (with the same unifiers) in which each pair has at least one flexible term. Hence we can traverse the first-order structure of terms to produce this effect. For a set consisting only of flexible-flexible pairs, a unifier is easily produced. For a flexible-rigid pair, Huet has shown that two kinds of substitutions are possible: the first makes the head of the flexible term "imitate" that of the rigid term, and the second "projects" one of its arguments in order that the head of the resulting term may be unified with the rigid term. Let $F = \lambda x_1.\ldots, \lambda x_n.(f\ t_1 \ldots t_k)$ and $R = \lambda x_1.\ldots.\lambda x_n, (c\ s_1 \ldots\ s_j)$ respectively be the flexible and rigid terms in head normal form; and let the type of f be $\alpha_1 \rightarrow \ldots \rightarrow \alpha_k \rightarrow \beta$. Then

1. If c is a constant, the imitation substitution is defined as:

   $$\{\langle f, \lambda w_1, \ldots, \lambda w_k, (c\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_j\ w_1\ \ldots\ w_k))\rangle\};$$

2. If α_i is of the form $\beta_1 \rightarrow \ldots \rightarrow \beta_l \rightarrow \beta$, the i^{th} projection substitution, for $1 \leq i \leq k$, is defined as:

   $$\{\langle f, \lambda w_1, \ldots, \lambda w_k, (w_i\ (h_1\ w_1\ \ldots\ w_k)\ \ldots\ (h_l\ w_1\ \ldots\ w_k))\rangle\}\}.$$

   where the h's above are new variables of appropriate types. Note that these substitutions are determined entirely by the heads of the flexible and rigid terms.

Unlike the unification of two first-order terms, for which there is a unique most general unifier (if one exists), higher-order unification can produce more than one maximally general unifier. It is even possible that two terms have countably infinite unifiers.

4.2.2 Example

With reference to the example from section 4.1, we want to find substitutions for `F1`, `F2`, `F3`, `F4`, `F5`, and `F7` so that `(F1 (F2 F4) (F3 F7 (F2 F5)))` = `(saw mike mary)`. The sequence of steps needed to find a unifier is shown below. The derivation is presented in terms of triples of the form `< T1 , T2 , Subst >`, where `T1` and `T2` the two higher-order terms to be unified, and `Subst` is the set of substitutions applied so far.

```
            < (F1 (F2 F4) (F3 F7 (F2 F5))) , (saw mike mary) , {} >
Projection:     { F1 <- U1\U2\(U1 (H1 U1 U2)) }
---->       < (F2 F4 (H1 (F2 F4) (F3 F7 (F2 F5)))) , (saw mike mary) ,
                  { F1 <- U1\U2\(U1 (H1 U1 U2)) } >
Projection:     { F2 <- U1\U2\(U2 (H2 U1 U2)) }
---->       < (H1 ... (F3 F7 (F2 F5)) (H2 F4 (H1 ... (F3 F7 (F2 F5))))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 (H1 U1 U2)),
                    F2 <- U1\U2\(U2 (H2 U1 U2)) } >
Projection:     { H1 <- U1\U2\U3\(U2 (H3 U1 U2 U3)) }
---->       < (F3 F7 (F2 F5) (H3 ... (F3 F7 (F2 F5)) (H2 F4 ...))) ,
```

```
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)) } >
Projection:     { F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) }
---->       < (F2 F5 (H4 F7 (F2 F5) (H3 ... ... (H2 F4 ...))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) } >
Apply substitution for F2:
---->       < (H4 F7 ... (H3 ... ... (H2 F4 ...)) (H2 F5 ...)) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 (H4 U1 U2 U3)) } >
Projection:     { H4 <- U1\U2\U3\U4\(U1 (H5 . . U3 .) (H6 . . . U4)) }
---->       < (F7 (H5 . . (H3 ... ... (H2 F4 ...)) .) (H6 . . . (H2 F5 ...))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4) (H6 U1 U2 U3 W4))) } >
Imitation:      { F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) }
---->       < (saw (K1 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .)))
                 (K2 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .)))) ,
                  (saw mike mary) ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4) (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) } >
Now unify first argument:
---->       < (K1 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .))) , mike ,
                  { F1 <- U1\U2\(U1 W3\(U2 (H3 U1 U2 W3))),
                    F2 <- U1\U2\(U2 (H2 U1 U2)),
                    F3 <- U1\U2\U3\(U2 W4\(U1 (H5 U1 U2 U3 W4) (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) } >
Projection:     { K1 <- U1\U2\U1 }
Projection:     { H5 <- U1\U2\U3\U4\U3 }
Projection:     { H3 <- U1\U2\U3\U4\U3 }
Projection:     { H2 <- U1\U2\U1 }
---->       < F4 , mike ,
                  { F1 <- U1\U2\(U1 W3\(U2 W3)),
                    F2 <- U1\U2\(U2 U1),
                    F3 <- U1\U2\U3\(U2 W4\(U1 U3 (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw U1 (K2 U1 U2)) } >
Imitation:      { F4 <- mike }
Unify second argument:
            < (K2 (H5 . . (H3 . . (H2 F4 .)) .) (H6 . . . (H2 F5 .))) , mary ,
                  { F1 <- U1\U2\(U1 W3\(U2 W3)),
                    F2 <- U1\U2\(U2 U1),
                    F3 <- U1\U2\U3\(U2 W4\(U1 U3 (H6 U1 U2 U3 W4))),
                    F7 <- U1\U2\(saw U1 (K2 U1 U2)),
                    F4 <- mike } >
```

```
Projection:     { K2 <- U1\U2\U2 }
Projection:     { H6 <- U1\U2\U3\U4\U4 }
Projection:     { H2 <- U1\U2\U1 }
---->      < F5 , mary ,
                  { F1 <- U1\U2\(U1 W3\(U2 W3)),
                    F2 <- U1\U2\(U2 U1),
                    F3 <- U1\U2\U3\(U2 W4\(U1 U3 W4)),
                    F7 <- U1\U2\(saw U1 U2),
                    F4 <- mike } >
Imitation:      { F5 <- mary }
```

Applying η-conversion to the substitutions for F1 and F3 we obtain the final set of substitutions:

```
{ F1 <- U1\U2\(U1 U2),
  F2 <- U1\U2\(U2 U1),
  F3 <- U1\U2\U3\(U2 (U1 U3)),
  F7 <- U1\U2\(saw U1 U2),
  F4 <- mike,
  F5 <- mary }
```

4.2.3 Efficiency Considerations

The above example illustrates that the search for a unifier occurs in a large space. However, it appears that a breadth-first exploration of substitutions in conjunction with the constraints imposed by multiple examples can help prune the search space quickly. To demonstrate this point, consider the example from section 4.2.2 and the following substitution:

```
           < (F1 (F3 F9) (F4 F12 (F3 F10))) , (saw mike mary) , {} >
Imitation:    { F1 <- U1\U2\(saw (K1 U1 U2) (K2 U1 U2)) }
```

which would be a legitimate substitution at this point. However, if the other examples are processed simultaneously, we would also have

```
< (F1 (F3 F9) (F4 F13 (F3 F10))) , (visited mike mary) , {} >
```

for which the above substitution for F1 would fail immediately.

Often certain fixed combinations of substitutions are performed, and hence it would be desirable to incorporate these combinations as "macros," so as to speed up the search for a solution. E.g., consider the substitution for F1 in the example above:

```
F1 <- U1\U2\(U1 (H1 U1 U2))
```

It is unnecessary in this case to introduce the additional complexity associated with the new variable H1. Instead in many cases we can assume that H1 will be bound to a projection like:

```
H1 <- U1\U2\U2
```

that is, F1 would simply be:

```
F1 <- U1\U2\(U1 U2)
```

eliminating a large section of the search space.

An additional reason for using a breadth-first search is to guarantee termination whenever there is a solution to the toplevel set of equations, since the higher-order unification procedure is only recursively enumerable. It is noteworthy that all equations have variables only on one side, and this should further help in constraining the search space. The search space can also be drastically reduced by providing the semantic representation for some of the terminal symbols, which should be straightforward in many domains. Finally, the knowledge of types of variables and function symbols will also constrain the search for a correct set of substitutions. The semantic representations given in the training instances are associated with types in the following way: each constant that is not a function constant is associated with the individual type **i**; each subterm representing an uninterpreted function constant and all its arguments (usually designated by a pair of parentheses) is associated with the boolean type **o**. The types of all the other subterms are inferred by the unification procedure.

5. Partial Execution and Reversibility

The higher-order DCGs that are constructed can be used for computing the semantics of a sentence quite efficiently, but not so efficiently for generating a sentence given its semantic representation. For example, if the higher-order rule $s((F\ A\ B)) \rightarrow np(A),\ vp(B)$ in the above grammar is used for parsing, the semantics A for np and B for vp are computed first, and then F is applied to A and B to obtain the semantics for s, whereas if used for generation, A and B would have to be assigned nondeterministically using higher-order unification.

The key to reversibility is to transform the higher-order rules into first-order rules where possible. When β-reduction amounts to simply instantiating variables according to the structure of the λ-terms, explicit β-reduction can be replaced by appropriately structuring the argument positions where the logical forms are returned. This technique is called *partial execution* (Pereira 1987).

A higher-order DCG rule is partially executed by recursively applying all functions to their arguments; if a function variable has not been instantiated but is applied to some argument, all of its occurrences in the rule are instantiated to an abstraction of the form `X\Y`, where `X` and `Y` are new variables, so that the application can be performed. We illustrate partial execution with the example of section 4.1. Considering the higher-order DCG constructed in section 4.1, rule (r1) can be partially executed in the following way:

```
     s((X\Y\(X Y) A B)) --> np(A), vp(B).
  =  s((A B)) --> np(A), vp(B).
  =  s((C\D B)) --> np(C\D), vp(B).
  =  s(D) --> np(B\D), vp(B).
```

In this partially executed form, the symbol `\` is simply an infix binary constructor, and therefore can now take structured terms in both of its argument positions.

Rule (r2) can be similarly converted:

```
   np((X\Y\(Y X) A)) --> pn(A).
=  np(Y\(Y A)) --> pn(A).
=  np((B\C)\(B\C A)) --> pn(A).
=  np((A\C)\C) --> pn(A).
```

Likewise rule (r3):

```
   vp((X\Y\Z\(Y (X Z)) A B)) --> tv(A), np(B).
=  vp(Z\(B (A Z))) --> tv(A), np(B).
=  vp(Z\(B (C\D Z))) --> tv(C\D), np(B).
=  vp(Z\(B D)) --> tv(Z\D), np(B).
=  vp(Z\(E\F D)) --> tv(Z\D), np(E\F).
=  vp(Z\F) --> tv(Z\D), np(D\F).
```

There are no applications in the semantic representations of the terminal symbols here, therefore the following first-order DCG is obtained:

```
(r1)   s(A)       --> np(B\A), vp(B).
(r2)   np((A\B)\B) --> pn(A).
(r3)   vp(A\B)    --> tv(A\C),  np(C\B).
(r4)   pn(mike)  --> [mike].
(r5)   pn(mary)  --> [mary].
(r6)   pn(john)  --> [john].
(r7)   tv(A\B\(saw A B )    --> [saw].
(r8)   tv(A\B\(visited A B) --> [visited].
```

We are planning to investigate this technique further to see how often, and for what kinds of grammars, it is applicable.

6. Conclusions and Further Work

The underlying thesis of this work is that it is possible to mechanically transform an unambiguous context-free grammar into a definite-clause grammar using a finite set of examples such that the resulting DCG correctly defines the semantics for all sentences of the CFG. This problem not only poses interesting technical challenges but also has potential applications; and, to the best of our knowledge, the problem remains unaddressed in the literature. By restricting semantic representations to the simply typed λ-calculus and employing the principles of compositionality and intensionality, we show that one can adopt the technique of higher-order unification to systematically explore the space of solutions. Unlike first-order DCGs, higher-order DCGs are not efficiently reversible, but they can often be made reversible through the use of partial execution, which effectively turns the higher-order rules into first-order rules where possible.

We note that our objective differs from those of Berwick (1985), Ishizaka (1990) and others, who are concerned with inferring a grammar from example sentences; rather, given the grammar, our objective is to infer the semantics of sentences from examples. Natural languages are of interest in our work since they are good examples of languages whose semantics require the use quantified terms and hence the full use of the typed λ-calculus. However, our work is not directly concerned with devising suitable semantics for natural languages; it is the user's responsibility to construct both the grammar as well as the semantic representation for typical sentences in the typed λ-calculus. Our work is concerned with that subset of natural languages that can be adequately described with CFGs and the typed λ-calculus.

For applications such as natural query languages, it seems feasible to describe the language with a context-free grammar and also to insist on sentences whose meanings have no ambiguity. However, the proposed techniques apply equally to CFGs that have been extended with additional arguments to control rule application, which makes them effectively context sensitive. These arguments can be used for example to enforce number or gender agreement. Such attributes can be specified in the lexicon (see appendix for an example). The process of synthesizing semantic representations is essentially independent of such additional context sensitive features, which only restrict the number of sentences that can be generated from a semantic representation.

Future work will include investigation of correctness issues under the assumptions of compositionality and intensionality. We have to pursue further the problem of efficient implementation of higher-order unification in order to allow applications to larger grammars. Efficient implementation of higher-order unification seems quite feasible in our problem domain if all constraints are exploited, e.g., by choosing appropriate training instances and by solving higher-order equations simultaneously. We also need to clarify under which conditions a higher-order DCG can be partially executed. Resolving these issues satisfactorily will allow us attack larger applications like natural language interfaces to data bases or knowledge bases, and machine translation systems.

References

Abramson, H. & Dahl, V. (1989) *Logic Grammars*, Springer-Verlag.

Anderson, J. R. (1977) "Induction of Augmented Transition Networks," *Cognitive Science*, 1, 125-157.

Anderson, J. R. (1981) "A theory of language acquisition based on general learning principles," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, p. 97-103, Vancouver, B. C., Canada.

Barzdin, J. M. (1977) "Inductive inference of automata, functions and programs," *Amer. Math. Soc. Transactions*, Vol. 109(2), p. 107-112.

Bauer, M. (1979) "Programming by Examples," *Artificial Intelligence*, 12, 1-21.

Berwick, Robert C. (1985) "The Acquisition of Syntactic Knowledge," MIT Press, Cambridge, Massachusetts.

Biermann, A. W., Guiho, G, and Kodratoff, Y. (eds.) (1984) *Automatic Program Construction Techniques*, Macmillan Publishing Company.

Blum, L., and Blum, M. (1975) "Toward a mathematical theory of inductive inference," *Inform. Control*, 28, p. 125-155.

Castaing, J., and Kodratoff, Y. (1984) "Theorem Proving By The Study Of Example Proof Traces or Theorem Proving By A Correct Theorem Statement," In: *Automatic Program Construction Techniques*, A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), p. 421-431, Macmillan Publishing Company.

Colmerauer, A. (1978) "Metamorphosis Grammars," In: L. Bolc (ed), Lecture Notes in Computer Science, Springer-Verlag, vol. 63, pp. 133-189.

Dahl, V., and Saint-Dizier, P. (eds) (1988) "Natural Language Understanding and Logic Programming II," North-Holland.

Dowty, David R., Wall, Robert E. and Peters, Stanley (1981) "Introduction to Montague Semantics," Dordrecht, Holland; Boston: D. Reidel Pub. Co.; Hingham, MA.

Dymetman, M., and Isabelle, P. (1990) "Grammar bidirectionality through controlled backward deduction", in *Logic and Logic Grammars for Language Processing*, Patrick Saint-Dizier and Stan Szpakowicz (eds.), Ellis Horwood, p. 275-293.

Gordon, M. J. C. (1988) *Programming Language Theory and its Implementation*, Prentice Hall, New York.

Huet, G. P. (1975) "A unification algorithm for typed λ-calculus," *Theoretical Computer Science*, 1 27-57.

Hutchins, W. J. (1986) "Machine translation: past, present, future," Ellis Horwood Limited.

Isabelle, P., Dymetman, M. and Macklovitch, E. (1988) "CRITTER: a translation system for agricultural market reports," *Proc. 12th International Conference on Computational Linguistics*, pp. 261-266. Budapest, August.

Ishizaka, H. (1990) "Polynomial Time Learnability of Simple Deterministic Languages," *Machine Learning*, 5, 151-164.

Jensen, D. C., and Pietrzykowski, T. (1976) "Mechanizing ω-Order Type Theory through Unification," *Theoretical Computer Science* 3, pp. 123-171.

Kodratoff, Y. (1979) "A class of functions synthesized from a finite number of examples and a LISP program scheme," *International Journal of Computer and Information Sciences*, 8, 489-521.

Landsbergen, J. (1987) *Montague Grammar and Machine Translation*, Eindhoven, Holland, Philips Research M.S. 14.026.

Lassez, J. L., Maher, M. J. and Marriott K. (1988) "Unification Revisited," in *Foundations of Deductive Databases and Logic Programming*, Jack Minker (ed.), Morgan Kaufmann, Los Altos, CA, pp. 587-625.

Lehnert, W. G. (1987) "Learning to Integrate Syntax and Semantics," Proceedings of the Fourth International Workshop on Machine Learning, Irvine, 1987, Morgan Kaufmann, Los Altos, CA, 179-190.

Michalski, R. S., Carbonell, J. G., and Mitchell, T. M. (1983), *Machine Learning: An Artificial Intelligence Approach*, vol. 1-3, (Los Altos, CA: Morgan Kaufmann).

Mitchell, T. M. (1978) "Version Spaces: An Approach to Concept Learning," Stanford, CA: Stanford University, (Ph.D. dissertation).

Nadathur, G., and Miller, D. (1988) "An Overview of λProlog," *Proc. 5th ICLP*, pp. 810-827.

Nadathur, G., and Jayaraman, B. (1989) "Towards a WAM Model for Lambda-Prolog," *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio.

Nirenburg, Sergei (1987) "Machine translation: Theoretical and methodological issues," Cambridge University

Press.

Pereira, F. C. N. (1990) "Semantic Interpretation as Higher-Order Deduction," *Proc. Second European Workshop on Logics and AI*, Amsterdam.

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.

Pereira, F. C. N. (1987) "Prolog and Natural-Language Analysis," CSLI.

Plotkin, G. D. (1970) "A Note on Inductive Generalization," *Machine Intelligence 5*, B. Meltzer and D. Michie (eds.), p. 153-163, American Elsevier, New York.

Plotkin, G. D. (1971) "A Further Note on Inductive Generalization," *Machine Intelligence 6*, B. Meltzer and D. Michie (eds.), p. 101-124, American Elsevier, New York.

Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the Association for Computing Machinery*, vol. 12, no. 1.

Schank, R. (1975) "Conceptual Information Processing," New York, NY: American Elsevier.

Sedlock, D. (1988) "Natural Language Generation in a Large Expert System," 8th International Workshop. Expert Systems and their Applications, Avignon, France, 1988. (Nanterre, France: EC 2), pp. 401-12, vol. 1.

Selfridge, M. (1981) "A computer model of child language acquisition," In: *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, p. 92-96, Vancouver, B. C., Canada.

Shapiro, S. C. (1979) "The SNePS Semantic Network Processing System," In *Associative Networks: Representation and Use of Knowledge by Computers*, N. V. Findler (ed), New York: Academic Press, 179-203.

Shavlik, J. W. (1990) "Acquiring Recursive and Iterative Concepts with Explanation-Based Learning," *Machine Learning*, 5, 39-70.

Shieber, S. M. (1986) "A uniform architecture for parsing and generation," *Proc. 12th International Conference on Computational Linguistics*, pp. 614-619, Budapest, August.

Smith, D. R. (1984) "The Synthesis of LISP Programs from Examples: A Survey," In: *Automatic Program Construction Techniques*, A. W. Biermann, G. Guiho, and Y. Kodratoff (eds.), pp. 307-324, Macmillan Publishing Company.

Summers, P. D. (1977) "A methodology for LISP program construction from examples," *Journal of the Association of Computing Machinery*, 24, 161-175.

Velardi, P. (1989) "Natural Language Interfaces to Databases: Features and Limitations," Entity-Relationship Approach: A Bridge to the User. Proceedings of the Seventh International Conference, Rome, Italy, 1988. (Amsterdam, Netherlands: North-Holland), pp. 35-40.

Vere, S. A. (1975) "Induction of Concepts in the Predicate Calculus," In: *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, USSR, p. 281-287.

Wallace, M. (1984) "Communicating with Databases in Natural Language," Ellis Horwood Services (A-1), European Computer-Industry Research Center GMBH.

Winston, P. H. (1975) "Learning Structural Descriptions from Examples," In: *The Psychology of Computer Vision*, P. H. Winston (ed.), New York: McGraw-Hill Book Company, 157-209.

Woods, W. A. (1970) "Transition network grammars for natural language analysis," *Communications of the ACM*, 13: 591-606.

Appendix: Sample Execution

Below is a CFG to be augmented with semantic interpretation rules.

```
s --> np, vp.
np --> det, n_rel.
np --> pn.
vp --> tv, np.
vp --> iv.
n_rel --> n, rel.
rel --> rel_pro, vp.
rel --> [].
pn --> [john].
pn --> [mary].
pn --> [mike].
det --> [a].
det --> [every].
n --> [girl].
n --> [boy].
iv --> [sings].
iv --> [plays].
tv --> [saw].
tv --> [visited].
rel_pro --> [that].
```

The training instances for this example are:

```
train(1,[john,sings],(sings john)).
train(2,[mary,sings],(sings mary)).
train(3,[john,plays],(plays john)).
train(4,[john,saw,mary],(saw john mary)).
train(5,[mike,saw,mary],(saw mike mary)).
train(6,[john,visited,mary],(visited john mary)).
train(7,[a,girl,sings],(exists X\(and (girl X) (sings X)))).
train(8,[every,girl,sings],(all X\(implies (girl X) (sings X)))).
train(9,[a,boy,sings],(exists X\(and (boy X) (sings X)))).
train(10,[john,saw,a,girl,that,sings],
         (exists X\(and (and (girl X) (sings X)) (saw john X)))).
```

Here is the augmented higher-order DCG in which each grammar symbol obtained an additional argument representing the semantics of the corresponding grammar constituent:

```
s((A\B\ (A B) C D)) --> np(C),vp(D).
np((A\B\C\ (B (A C)) D E)) --> det(D),n_rel(E).
np((A\B\ (B A) C)) --> pn(C).
vp((A\B\C\ (B  (A C)) D E)) --> tv(D),np(E).
vp((A\B\ (A B) C)) --> iv(C).
n_rel((A\B\C\ (C (B A)) D E)) --> n(D),rel(E).
rel((A\B\C\D\ (A C B D) E F)) --> rel_pro(E),vp(F).
rel(A\B\ (A B)) --> [].
pn(john) --> [john].
pn(mary) --> [mary].
pn(mike) --> [mike].
det(A\B\ (exists C\ (and (B C) (A C)))) --> [a].
```

```
det(A\B\ (all C\ (implies (B C) (A C)))) --> [every].
n(A\ (girl A)) --> [girl].
n(A\ (boy A)) --> [boy].
iv(A\ (sings A)) --> [sings].
iv(A\ (plays A)) --> [plays].
tv(A\B\ (saw A B)) --> [saw].
tv(A\B\ (visited A B)) --> [visited].
rel_pro(A\B\C\ (and (A C) (B C))) --> [that].
```

After partial execution and generalization of the representations of terminals, where possible, the following first-order DCG is obtained.

```
s(A) --> np(D\A),vp(D).
np(A\B) --> det(A\E),n_rel(E\B).
np((A\B)\B) --> pn(A).
vp(A\B) --> tv(A\E),np(E\B).
vp(A\B) --> iv(A\B).
n_rel((A\B)\B) --> n(E),rel(E\A).
rel(A\B\C) --> rel_pro(A\F\B\C),vp(F).
rel((A\B)\A\B) --> [].
pn(A) --> [A],{lex(A,pn,D)}.
det((A\B)\ (A\C)\ (exists A\ (and C B))) --> [a].
det((A\B)\ (A\C)\ (all A\ (implies C B))) --> [every].
n(A\ (B A)) --> [B],{lex(B,n,E)}.
iv(A\ (B A)) --> [B],{lex(B,iv,E)}.
tv(A\B\ (C A B)) --> [C],{lex(C,tv,F)}.
rel_pro((A\B)\ (A\C)\A\ (and B C)) --> [that].
```

For some of the lexical categories in this example it is possible to systematically construct the semantic representations from the words themselves. E.g., if `TV` is a transitive verb in past tense, its representation is always of the form `A\B\(TV A B)`. Therefore this construction can be generalized to all transitive verbs in past tense. The lexicon used for this example is listed below. Each entry is of the form `lex(Word,Category,Attributes)`. Note, however, that in general there doesn't need to be any resemblance between a word and its semantic representation in order for our system to infer the representations of those words from the examples. The additional generalizations are performed only if there exist such regularities.

```
lex( a, det, [_,sing|_] ).
lex( every, det, [_,sing|_] ).

lex( that, rel_pro, _ ).
lex( which, rel_pro, [_,_,neut|_] ).

lex( being, n, [_,sing,neut|_] ).
lex( book, n, [_,sing,neut|_] ).
lex( boy, n, [_,sing,masc|_] ).
lex( child, n, [_,sing,neut|_] ).
lex( girl, n, [_,sing,fem|_] ).
lex( person, n, [_,sing|_] ).
lex( picture, n, [_,sing,neut|_] ).
lex( professor, n, [_,sing|_] ).
lex( program, n, [_,sing,neut|_] ).
lex( pupil, n, [_,sing|_] ).
lex( student, n, [_,sing|_] ).

lex( bertrand, pn, [_,sing,masc|_] ).
lex( eliza, pn, [_,sing,fem|_] ).
lex( john, pn, [_,sing,masc|_] ).
lex( mary, pn, [_,sing,fem|_] ).
```

```
lex( mike, pn, [_,sing,masc|_] ).
lex( nancy, pn, [_,sing,fem|_] ).
lex( principia, pn, [_,sing|_] ).
lex( shrdlu, pn, [_,sing,neut|_] ).
lex( terry, pn, [_,sing|_] ).

lex( drew, tv, [past|_] ).
lex( ran, tv, [past|_] ).
lex( read, tv, _ ).
lex( saw, tv, [past|_] ).
lex( studied, tv, [past|_] ).
lex( taught, tv, [past|_] ).
lex( visited, tv, [past|_] ).
lex( wrote, tv, [past|_] ).

lex( halts, iv, [pres,sing|_] ).
lex( laughs, iv, [pres,sing|_] ).
lex( learns, iv, [pres,sing|_] ).
lex( plays, iv, [pres,sing|_] ).
lex( runs, iv, [pres,sing|_] ).
lex( sings, iv, [pres,sing|_] ).
lex( talks, iv, [pres,sing|_] ).
lex( works, iv, [pres,sing|_] ).

lex( red, adj, _ ).
lex( blue, adj, _ ).
lex( good, adj, _ ).
lex( bad, adj, _ ).
lex( smart, adj, _ ).
lex( little, adj, _ ).
lex( big, adj, _ ).
```

Below are some sample executions of the grammar constructed above, demonstrating features like recursive rule applications for relative clauses, quantified noun phrases, and reversibility. The system has also been successfully tested on more complex grammars that include adjectives and passive voice constructions.

```
|: john saw mary.                                     % Natural language input.

   (saw john mary).                                   % Semantic representation computed
                                                      % by the grammar.
|: john wrote a program that talks.

   (exists A\ (and (and (program A) (talks A)) (wrote john A)).

|: john taught every student that wrote a program that drew a picture.

   (all A\ (implies  (and  (student A)
                           (exists B\ (and  (and  (program B)
                                                  (exists C\ (and  (picture C)
                                                                   (drew B C))))
                                            (wrote A B))))
                     (taught john A))).


|: (saw john mary).                                   % For generation, the semantic representation
                                                      % is the input.
   john saw mary.                                     % Sentence generated by the grammar.

|: (exists A\ (and  (and  (student A)  (read A principia))  (visited A john))).

   a student that read principia visited john.
```

```
|: (all A\ (implies  (and  (student A)
                           (exists B\ (and  (and  (program B)
                                                  (exists C\ (and  (picture C)
                                                                   (drew B C))))
                                            (wrote A B))))
                     (taught john A))).

   john taught every student that wrote a program that drew a picture.
```