Using Bidirectional Semantic Rules for Generation

Jim Barnett and **Inderjeet Mani**
Microelectronics and Computer Technology Corporation (MCC)
3500 West Balcones Center Drive
Austin, Texas 78759
U.S.A.

Abstract

This paper describes the use of a system of semantic rules to generate noun compounds, vague or polysemous words, and cases of metonymy. The rules are bidirectional and are used by the understanding system to interpret the same constructions.

Introduction

In generation systems that are paired with understanding systems, bidirectionality is desirable for reasons that are both theoretical (a single model of linguistic behaviour) and practical (shorter development time, greater consistency, etc.)[1]. Recently, [Shieber et al. 89] and [Calder at al. 89] have presented generation algorithms that share both semantics and syntax with the understanding system. This paper presents an extension of these algorithms to deal with phenomena that have often been lumped together under 'pragmatics', namely noun compounding, metonymy (the use of a word to refer to a related concept), and vague or polysemous words like "have."

The difficulty with these constructions is that they are productive, and cannot be handled easily by simply listing meanings in a lexicon. Taking noun compounding as an example, we have "corn oil" and "olive oil" referring to oil made from corn or olives. We could add a lexical sense for "corn" meaning "made from corn," but then we face an explosion in the size of the lexicon, and an inability to understand or generate novel compounds: if we acquire "safflower" as the name of a plant, we would like the system to be able to handle "safflower oil" immediately, but this won't be possible if we need a separate lexical sense to handle compounding. The system will be more robust (and the lexicon more compact) if we can derive the desired sense of "safflower" from the basic noun sense when we need it. We have therefore developed a system of bidirectional *semantic rules* to handle these phenomena at the appropriate level of generality.

[1]For more detailed arguments along these lines, see [Appelt 87], [Shieber 88], [Jacobs 88a].

We have implemented these rules in Common Lisp as part of the KBNL system [Barnett et al. 90] at MCC, but nothing depends on the idiosyncracies of our formalisms or implementation, so the technique is compatible with a wide variety of theories of the ***kinds*** of relations that are likely to occur in these constructions, as in, e.g., [Finin 80] for noun compounds and [Nunberg 78] for oblique reference.

The Framework

The algorithms for recognition and generation use an agenda-based blackboard for communication and control [Cohen et al. 89]. Our syntax component uses an extension of Categorial Unification Grammar [Wittenburg 86] as the phrase-structure component of an LFG-style functional representation (f-structure), and the semantic component maps from this representation to sets of assertions in the interface language of the CYC knowledge base [Lenat et al. 90].

Semantic rules map partial semantic interpretations onto other partial interpretations. They consist of a left-hand side and a right-hand side, each consisting of one or more templates, plus a mechanism for mapping an instantiation of either set of templates onto an instantiation of the other set. The intuitive semantics of these rules is that any interpretation that matches the left-hand side licenses a second interpretation matching the right-hand side. For example, we can use the name of an author to refer to his works ("I read Shakespeare"), and the corresponding semantic rule states that the existence of an NP denoting an artist licences the use of the same NP to refer to his works. The generation system applies the rules in a backward-chaining direction, while the understanding system runs them forward. A later section contains a fuller discussion of the implementation of the rules, while the next sections discuss their use at runtime.

Generation

The generator is divided into strategic and tactical components. The former takes a frame as input and creates a description of it based on a set of discriminative properties which are recorded in the KB and indicate which aspects of a frame are likely to be salient. If a comparison class is available, the resulting description uniquely identifies the frame with respect to that class, otherwise it contains default 'interesting' properties. Once it has generated this set of assertions, the strategic component calls the tactical component with a goal *Semantics : Syntax*, where *Semantics* consists of the assertions plus the distinguished variable that the utterance is 'about', and *Syntax* is an f-structure (which may specify no more than the category.)

Given this input, the tactical component uses a variant of the semantic-head driven algorithms described by [Calder at al. 89] and [Shieber et al. 89] to generate a phrase whose syntax and semantics match the goal. Before examining this algorithm, we note that in categorial grammars, most of the syntactic information is contained in the lexical definitions of words. For example, the lexical entry for a transitive verb like "read" specifies that it takes an object NP to its right and then a subject NP to its left. Any such constituent that takes at least one argument is called a *functor*, while a constituent with no arguments is called *atomic*. Functors and their arguments are combined by a small number of ***binary rules***, and there is also a set of ***unary rules***, which can change the category of a constituent (forming passive verbs out of actives, for example.)

Next we define two relationships between constituents and goals: first, a constituent *matches* a goal if its semantics subsumes the goal's semantics and its syntactic category is the same as the goal's, with possible extra arguments. Thus the transitive verb "eat", with category $S\backslash NP/NP$, is a syntactic match for the goal category S because it will be an S once it gets its arguments. Second, a constituent *satisfies* a goal if it has identical semantics and its f-structure is a supergraph of the goal's f-structure.

Given this syntactic framework, the algorithm works by peeling off lexical functors and recursing on their arguments until it bottoms out in an atomic constituent. Given a goal, the first step consists of lexical look-up to find an item that matches the goal. Once this item, called the semantic head, is found, the algorithm proceeds both top-down and bottom up. If the semantic head is a functor, it proceeds top-down trying to solve the sub-goal for its argument. Once this sub-goal is satisfied, the algorithm works bottom-up by applying unary grammar rules to to the argument constituent alone, or binary rules to combine it with the functor. When a complete constituent is found which satisfies the goal, we are done.

Extension: Goal Revision

The algorithm described above assumes a fixed set of choices in the lexicon. It can generate metonymic expressions and noun compounds, but only at the cost of massive lexical ambiguity. We therefore extend it by considering the possibility of goal revision as an alternative to the lexical look-up step[2]. By running a semantic rule backward, we can map the current goal onto one or more new goals to which the algorithm recursively applies. Satisfying the new goals will generate an expression with the desired meaning and thus indirectly satisfy the original goal.

Revision using noun compounding rules leads to a binary decomposition of the original goal, as shown in Figure 2, while metonymy rules result in a unary decomposition, as shown in Figure 3. From this perspective, we note that the lexical look-up of a functor can be viewed as a kind of guided binary decomposition (Figure 1), splitting the original goal into two sub-goals with the knowledge that one of them will be satisfied immediately.

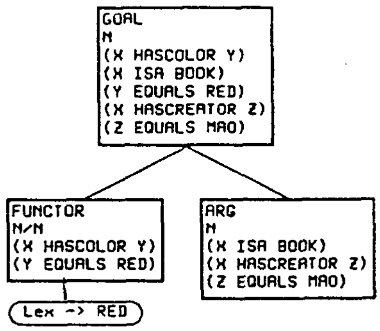

Figure 1: Lexical Lookup as Decomposition

Our extension to the algorithms of [Calder at al. 89] and [Shieber et al. 89] thus amounts to the decision to allow top-down decomposition to be guided by rules as well as lexical items. As we would expect, this is the mirror image of the situation during understanding, where semantic rules are used as an extension to the lexicon in the process of merging translations bottom-up. The extended algorithm is shown in the Appendix.

Controlling Rule Application

The strategic component can control the choice among alternatives through its specification of the goal's syntax. For example, the strategic component can force the use of a compound by providing an appropriately detailed f-structure (i.e., one that specifies the presence of a modifier of category N.) If it does so, no matter whether we are in best-first or all-paths mode, only the compounding alternative will succeed and satisfy the syntactic goal. On the other hand, if the syntactic goal is underspecified, the output (in best-first mode) will

[2]The notion of goal revision in generation dates back to [Appelt 83] where various conditions could lead to replanning of the input; for recent work incorporating goal revision see [Vaughan et al. 86].

depend on the tactical component's heuristic ordering. In this case, given the default ordering which prefers lexical look-up to noun compounding to metonymy, the tactical component will use a noun compound when lexical look-up fails (i.e., there is no corresponding adjective or preposition). Another result of this default ordering is that metonymy will never fire in the absence of a syntactic specification since there is always another way (unless the lexicon is incomplete) of saying the same thing using words that are in the lexicon. However, the literal alternative is usually more verbose than the metonymous expression, so the strategic component can force the use of metonymy by specifying a limit on the number of words the tactical component is allowed to use. Given a limit of 3 words, descriptive phrases like "a book by Joyce" will fail, and only the metonymous expression "Joyce" will succeed.

In best-first mode, substantial improvements in efficiency are possible by re-ordering the alternatives based on the syntactic properties of the goal. For example, it makes sense to try metonymy first if the desired length is significantly less than the number of assertions in the goal's semantics, since each lexical item normally covers only a few assertions.

Generating Noun Compounds

Suppose we have a **Software-Machine** rule, stating that if "y" denotes any kind of *Software* and "x" a computer, "a y x" means a *Computer* x that *CanRunLanguage* y. Now consider a goal with semantics *(W ISA Computer)(Z Equals Lisp)(W CanRunLanguage Z)*, distinguished variable *W*, and syntax NP. There is no lexical item covering all these assertions, or any lexical functor covering part of them (i.e., "Lisp" is not in the lexicon as an adjective.) Thus, even in best-first mode, we will end up applying the **Software-Machine** rule to this goal, resulting in the decomposition shown in Figure 2.

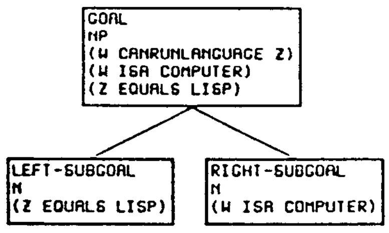

Figure 2: Decomposition for a Noun Compound

Recursing on the sub-goals, the lexical item "Lisp" will satisfy the left sub-goal, and "machine" will satisfy the other. Combining the sub-goal solutions yields "Lisp machine" as a solution to the original goal.

Multiple compounds are handled by repeated invocations of the rules. Suppose we have a **Mechanism-Maintenance** rule, stating that if "x" denotes a *Machine* and "y" denotes any kind of *MaintenanceOperation*, "x y" denotes a *MaintenanceOperation* y with y *Maintains* x. Given input semantics *(Y ISA RepairOperation)(Y Maintains X)(X ISA Computer)(X CanRunLanguage Z)(Z Equals Lisp)*, with distinguished referent *Y*, the maintenance rule will eventually fire, generating patterns for a head *(Y ISA RepairOperation)* and a modifier *(X ISA Computer)(X CanRunLanguage Z)(Z Equals Lisp)*. The head's goal will be satisfied by the entry for "repair", but processing of the modifier will invoke the **Software-Machine** rule, just as in the example above. The output will be "Lisp machine repair", with the left-branching structure [[Lisp machine] repair].

For an example of a right-branching compound, suppose we have a **Product-Manufacturer** rule stating that if "x" is the name of a *Product* and "y" is the name of a *Company*, then "a y x" is a *Product* x that is *ManufacturedBy* company y. Given the input *(X ISA Computer)(X ManufacturedBy Y)(X CanRunLanguage Z)(Y Equals Symbolics)(Z Equals Lisp)*, the product rule will fire, producing a modifier sub-goal for *(Y Equals Symbolics)* and a head sub-goal for *(X ISA Computer)(X CanRunLanguage Z)(Z Equals Lisp)*. This time the **Software-Machine** rule will be invoked on the head sub-goal, while lexical item "Symbolics" will satisfy the modifier sub-goal, and the output will be [Symbolics [Lisp machine]].

Generating Metonymic References

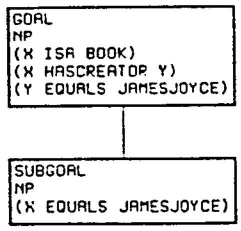

Figure 3: Decomposition for Metonymy

Suppose we have an **Artist** rule licensing the use of an artist's name to refer to his works, and are trying to generate an NP with semantics *(X ISA Book)(X HasCreator JamesJoyce)*. If we are in all-paths mode, or if the strategic component has requested a succinct solution, the **Artist** rule will fire, yielding the decomposition shown in Figure 3.

In this case, the new goal is immediately satisfied by the lexical entry for "Joyce", but in principle the solution to the metonymous sub-goal could involve peeling off various modifiers, yielding, for example, "an excruciatingly boring Walter Scott."

Semantic Rules during Understanding

The use of semantic rules during understanding is straightforward. Noun compounding rules are invoked in the appropriate syntactic context, and they merge the interpretations of the two nouns into a single meaning. Metonymy rules are invoked when the knowledge base rejects the literal meaning as inconsistent. Given input "Bill reads Shakespeare", the KB will not permit Bill to read the *person* Shakespeare, and the first try at semantic interpretation fails. We then invoke all available metonymy rules (on both "reads" and "Shakespeare"), and the **Artist** rule mentioned above succeeds, leading to the desired interpretation.

A hierarchy of rules is established by assigning each rule a level of effort, so that it is invoked only when the appropriate level is reached. Processing stops at the lowest level at which it can find an interpretation that the KB accepts. At each level, we consider all combinations of rules and lexical items from that level with items from previous levels, so it is possible for, e.g., a level 2 metonymy rule to feed a level 1 noun compounding rule, as in the example below, but only if there is no consistent interpretation at level 1 alone. The following trace shows the level 2 **Artist** metonymy rule given above interleaving with the level 1 **WorkPerformed** compounding rule (which states that if "x" denotes an instance of *WorkOfArt* and "y" denotes an instance of *Performance*, then "an x y" denotes a performance whose *WorkPerformed* is x.

```
Parse "a Bach recital"
...Level 1
...Trying WorkPerformed CPD Rule
...Failing
;; since JSBach is not a WorkOfArt
...Level 2
...Trying Artist Metonymy Rule on
 (X Equals JSBach)
...Success
;; now "Bach" ->
 (X ISA WorkOfArt)(X HasCreator JSBach)
...Level 1
...Trying WorkPerformed CPD Rule
...Success
;; interpretation is now ->
 (X ISA WorkOfArt)(X HasCreator JSBach)
 (Y ISA Performance)(Y WorkPerformed X)
```

Semantic Rule Implementation

Semantic rules, as we noted earlier, map partial semantic interpretations (sets of assertions) onto other partial interpretations[3]. The left and right hand sides of the rules are therefore specified as templates, i.e., as sets of assertions with variables in predicate and term positions. For both forward and backward chaining, rule application proceeds in three steps: unification with the rule's input templates, computation of the output bindings from the input bindings, and instantiation of the output template. The only difference between forward and backward chaining applications is which sides of the rule serve as the input and output. Rule application can fail at either of the first two steps if the input fails to match the specifications.

The only point worth mentioning about the unification and instantiation operations is that they allow sub-classes in the input to match super-classes in the rule pattern. Thus in the **MechanismMaintenance** example above, the classes *Computer* and *RepairOperation* in the input match *Machine* and *MaintenanceOperation* in the rule.

The complexity lies in the second step, where we must map a set of bindings for either side of the rule onto a set of bindings for the other side. Crucially, the values of bindings are either variables, or, most important, objects in the KB (classes or relations), and the second stage of rule application consists of 1) checking that the KB objects meet certain conditions, and 2) using the KB objects to compute the value of the new bindings. For example, when we apply the **Artist** metonymy rule to an expression in the forward direction, we must 1) check that it denotes a person, and 2) find the kinds of works of art that the person creates - poems, music, etc., since these are the types of objects that the name can be used to refer to.

To perform these operations, we create two kinds of expressions which can be evaluated in a binding environment: 1) Filter expressions, which have a single parameter, and return T or Nil depending on whether the binding of that parameter satisfies the expression 2) Bind expressions, which have two parameters, and return a set of bindings for the first parameter, based on evaluating the rest of the expression in the environment. Thus evaluating *(Filter (X ISA Dog))* in an environment will return T iff *X* is bound to an instance of *Dog*, and evaluating *(Bind Z (X CreatorOf))* will return a set of extended bindings, each with *Z* bound to one of the things that *X* is the *CreatorOf*. To apply a rule in either direction, we start with the bindings from the input unification (if it succeeded) and evaluate the Filter and Bind expressions in order, eliminating any environment in which a Filter expression returns Nil. The rule succeeds if it outputs one or more extended environments, which are then used to instantiate the output templates.

A rule's Filter and Bind statements compute a *relation* between input and output bindings. To reverse the

[3]The rules also allow for optional specification of syntactic categories and surface string.

rule, we need to compute the inverse of the relation.[4] To do this we reverse the order of the statements and replace each Bind statement (which computes a relation between variable bindings) with its inverse. Here we rely on the fact that the CYC KB automatically maintains inverses for all relations defined in it (for KBs without this feature, we would have to define inverses for all relations used in rules). If *CreatorOf* is the inverse of *HasCreator*, then the inverse of *(Bind Z (X CreatorOf))* is simply *(Bind X (Z HasCreator))*.

A few more details are necessary to complete the implementation. First, we define the relation *Instances*, which maps from classes to their elements, as the inverse of *ISA*. Next we define a concatenation operator *+* on relations, so that *(Z (Instances + HasCreator))* returns all the creators of instances of the class *Z*.[5] Next, we stipulate that if the variable *X* is already bound, *(Bind X (Z HasCreator))* acts as a Filter, returning T iff *X* is among the values for *(Z HasCreator.)* Finally, for reasons of efficiency, we cache separate patterns for backward and forward application, instead of reversing the expressions at run time. The generation and understanding patterns for the **Artist** rule are given below:

```
Input Pattern (W ISA Z)(new-var HasCreator X)
Bind          Y (Z Instances + HasCreator)
Filter        (Y ISA Person)
OutputPattern (X Equals Y)

Input Pattern (X Equals Y)
Filter        (Y ISA Person)
Bind          Z (Y CreatorOf + ISA)
Output Pattern(W ISA Z)(new-var HasCreator Y)
```

Running the rule backward on *(Q ISA Book)(Q HasCreator JamesJoyce)*, initial unification binds *Q* to *W*, *Z* to *Book* and *Y* to *JamesJoyce*. The Bind expression serves as a filter in this case, checking that *Y* is in fact the *CreatorOf* some instance of *Z*, the Filter expression checks that *Y* is a *Person*, and the output is *(X Equals JamesJoyce)*, which will match the lexical semantics for "Joyce" permitting us to use that NP to refer to the book. Running the rule forward on *(X Equals JamesJoyce)*, *Y* is bound to *JamesJoyce*, the Filter expression again checks that *Y* is a *Person*, and the Bind expression binds *Z* to all the classes of objects *Y* is the *CreatorOf*(in this case, the class *Book*). The output is an expression denoting any *Book* which *HasCreator JamesJoyce*, thus letting us understand "Joyce" as referring to a *Book*.

The rule format is similar for noun compounds, except that there are two input patterns (in the forward direction) and a single output pattern. During understanding, the two patterns are unified with the interpretations of the head and modifier nouns, the Filters and Binds work as before, and the output pattern is the interpretation of the compound. Backward application is as before, except that the output is a pair of instantiated patterns which the generation routine then uses as new goals.

[4]Mathematically, a relation is a set of ordered pairs, and its inverse is the set of inverted pairs. Any relation is therefore guaranteed to have a unique inverse.

[5]The inverse of (r1 + r2) is ((inverse r2) + (inverse r1)).

Related Work

A variety of systems have used rules of the kind we are considering, either explicitly or implicitly, for use in ***understanding*** compounds, vague expressions, and metonymy, for example, [Dahl et al. 87], [Hobbs et al. 88], [Grosz et al. 85], [Stallard 87], but no mention is made of reversing these rules for generation.

A number of systems generate compounds, but most apparently do so using either phrasal lexicons (e.g., [Hovy 88], [Jacobs 88b], [Wilensky 88]), or multiple lexical senses (e.g., [Pustejovsky et al. 87], [Nirenburg et al. 88]), rather than rules of the sort we propose. Other generation systems apparently construct noun compounds via specialized (uni-directional) strategies specified in the interface to the tactical component [McDonald et al. 88], or a combination of these techniques [McKeown 82]. We have been unable to find discussions of generation of metonymy, though at least some cases of it could obviously be handled via lexical ambiguity.

Discussion

The use of semantic rules seems to us to handle most of the technical problems in providing an economical, bidirectional treatment of a variety of non-literal and/or vague constructions. At the heart of any reversible system is the notion of being able to run mappings forwards or backwards, so that, for example, the understanding and generation lexicons are inverses of each other. These rules are a natural extension of this mechanism to more complex constructions. Furthermore, these rules can handle a wide variety of phenomena. In addition to the examples discussed above, we have used semantic rules for the lexical semantics of vague words like "have" and "of", which are like noun compounding and metonymy in that the only alternative to massive lexical ambiguity is to compute the nature of the relation based on the interpretation of the arguments. This use of semantic rules for lexical semantics amounts to permitting a lexical item to further decompose its subgoal, instead of satisfying it immediately in the manner of Figure 1. Finally, these rules do not commit us to any particular analysis of the constructions in question (except insofar as they assume separate levels of syntactic and semantic representation.) To take noun compounding as an example, we can implement a wide variety of theories of the kinds of relations compounds express and of the hierarchies among them.

The main open issue is the development of strategies for the use of these expressions. The problem is most acute in the case of metonymy. At present, the strategic component can force the use of metonymous expressions by requiring brevity. However, use of metonymy is not just a matter of succinctness since it also tends to indicate informality and familiarity. In more extreme uses, it may have a poetic or humorous force. To use metonymy, compounds, or other vague expressions successfully, we need a theory of how they effect the discourse, as well as a strategic component which is sophisticated enough to exploit the theory. As a first step in this direction, we are implementing a discourse module which will allow us to address some of these issues. For example, metonymous expressions are safer when used to refer to classes of objects that have already been described than when used to introduce new ones. If "an Orvis fishing rod" has already been mentioned, then "an Orvis" is likely to make sense, even to people who wouldn't have understood it the first time around. However, such individual heuristics will be of limited usefulness until they are integrated into a comprehensive model of communication.

Acknowledgments

We are grateful to Elaine Rich for comments on an earlier draft of this paper.

Appendix: Generation Algorithm

The psuedocode for the generation algorithm is shown below, identifying the point of departure from the [Calder at al. 89] algorithm. The lexical lookup-step of line 1 is replaced with the more general top-down step of line 1a, by calling the new function *generate-tp-dn*. The rest of the (pseudo)code remains unchanged.

Here are the language constructs used in the pseudocode. We denote local variable assignment as $X := Y$, with scope extending to the immediate containing construct. Destructuring by pattern matching is allowed, e.g. $< X1\ X2\ X3 > := Y$ simultaneously binds *X1*, *X2* and *X3* to the corresponding components in *Y*. *AND* and *OR* have exactly the behavior of Common Lisp AND and OR.

For the sake of conciseness,the function *choose* is used as a shorthand for control strategies: in all-paths mode, it finds all solutions; in best-first mode it imposes a heuristic ordering on the choices and finds a single solution, finding any subsequent solutions on backtracking. The function *choose-tp-dn-operation* heuristically picks the best operation based on the goal. The functions *match* and *satisfy* are as defined earlier. The functions *apply-unary-bup-rule* and *apply-binary-bup-rule* constitute the rule application interface to grammar rules; similarly, the functions *apply-unary-tp-dn-rule* and *apply-binary-tp-dn-rule* constitute the rule application interface for the semantic rules.

```
FUNCTION generate(Goal);
1 AND(;;OLD: Subgoal := lex-decomp(Goal)
     ;; NEW:
1a    Subgoal := choose(generate-tp-dn(Goal))
     ;; AS BEFORE:
2     choose(generate-bup(Subgoal, Goal))).

FUNCTION generate-tp-dn(Goal);
 operation :=
  choose(choose-tp-dn-operation(Goal))
 CASE operation
   :lex
     lex-decomp(Goal)
   :unary-decomp
;; e.g. METONYMY RULES:
     choose(apply-unary-tp-dn-rule(Goal))
   :binary-decomp
;; e.g. COMPOUNDING/VAGUE WORD RULES:
     AND(<left-subgoal, right-subgoal> :=
         choose(apply-binary-tp-dn-rule
         (Goal)),
         choose(generate(left-subgoal)),
         choose(generate(right-subgoal)),
         choose(apply-binary-bup-rule
         (left-subgoal, right-subgoal))).

FUNCTION generate-bup(Subgoal, Goal);
 OR(satisfies(Subgoal, Goal),
    AND(Arg :=
 choose(extract-arg(Subgoal)),
        Goal1 :=
         choose(apply-binary-bup-rule
         (Subgoal, Arg)),
        choose(generate(Arg)),
        choose(generate-bup
        (Goal1, Goal))),
    AND(Goal1 :=
        choose(apply-unary-bup-rule
        (Subgoal)),
        choose(generate-bup
        (Goal1, Goal)))).

FUNCTION lex-decomp(Goal);
 AND(Subgoal :=
      choose(lexical-lookup(Goal)),
     matches(Subgoal, Goal),
     Subgoal).
```

References

[Appelt 83] D. E. Appelt, "Telegram: A Grammar Formalism For Language Planning", *Proceedings of the ACL*, M.I.T., 15-17 June, 1983.

[Appelt 87] D. E. Appelt, "Bidirectional Grammars and the Design of Natural Language Generation Systems", *TINLAP-3 Position Papers*, New Mexico State University, 7-9 January, 1987.

[Barnett et al. 90] J. Barnett, K. Knight, I. Mani, and E. Rich, "Knowledge and Natural Language Processing", to appear in *Communications of the ACM*, August, 1990.

[Calder at al. 89] J. Calder, M. Reape, and H. Zeevat, "An Algorithm for Generation in Unification Categorial Grammar", *Proceedings of the 4th Conference of the European Chapter of the ACL*, pp. 233-240, Manchester, 10-12 April, 1989.

[Cohen et al. 89] R. M. Cohen, T. P. McCandless, and E. Rich, "A Problem Solving Approach to Human-Computer Interface Management", *MCC Tech Report ACT-HI-306-89*, Fall 1989.

[Dahl et al. 87] D. Dahl, M. Palmer, and R. Passonneau, "Nominalizations in Pundit", *Proceedings of the ACL*, Stanford, 6-9 July, 1987.

[Finin 80] T. Finin, "The Semantic Interpretation of Compound Nominals", *University of Illinois*, Ph.D. Dissertation, 1980.

[Grosz et al. 85] B. Grosz, D. Appelt, P. Martin, and F. C. N. Pereira, "Team: An Experiment in the Design of Transportable Natural-Language Interfaces", *Artificial Intelligence.*

[Hobbs et al. 88] J. R. Hobbs, M. Stickel, P. Martin, and D. Edwards, "Interpretation as Abduction", *Proceedings of the ACL*, Buffalo, 7-10 June, 1988.

[Hovy 88] E. H. Hovy, "Generating Language with a Phrasal Lexicon", in D. D. McDonald and L. Bolc, eds., *Natural Language Generation Systems*, Springer-Verlag, 1988.

[Jacobs 88a] P. S. Jacobs, "Achieving Bidirectionality", *Proceedings of the 1th International Conference on Computational Linguistics*, Budapest, 22-27 August, 1988, pp. 267-269.

[Jacobs 88b] P. S. Jacobs, "PHRED: A Generator for Natural Language Interfaces", in D. D. McDonald and L. Bolc, eds., *Natural Language Generation Systems*, Springer-Verlag, 1988.

[Lenat et al. 90] D. Lenat and R. Guha, "Building Large Knowledge Based Systems, Representations and Inference in the CYC Project", *Addison Wesley*, 1990.

[McDonald et al. 88] D. D. McDonald and M. W. Meteer, "From Water to Wine: Generating Natural Language Text from Today's Application Programs", *Second Conference on Applied Natural Language Processing*, Austin, 9-12 February, 1988.

[McKeown 82] K. R. McKeown, "Generating Natural Language Text in Response to Questions About Database Structure", *University of Pennsylvania*, Ph.D. Dissertation, 1982.

[Nirenburg et al. 88] S. Nirenburg, R. McCardell, E. Nyberg, P. Werner, S. Huffman, E. Kenschaft and I. Nirenburg, "DIOGENES-88", *CMU Tech Report CMU-CMT-88-107*, June 1988.

[Nunberg 78] G. Nunberg, "The Pragmatics of Reference", *City College of New York*, Ph.D. Dissertation, 1978.

[Pustejovsky et al. 87] J. Pustejovsky and S. Nirenburg, "Lexical Selection in the Process of Language Generation", *Proceedings of the ACL*, Stanford, 6-9 July, 1987.

[Shieber 88] S. M. Shieber, "A Uniform Architecture for Parsing and Generation", *Proceedings of the 12th International Conference on Computational Linguistics*, Budapest, 22-27 August, 1988, pp. 614-619.

[Shieber et al. 89] S. M. Shieber, G. van Noord, R. C. Moore, and F. C. N. Pereira, "A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms", *Proceedings of the ACL*, Vancouver, 26-29 June, 1989.

[Stallard 87] D. Stallard, "The Logical Analysis of Lexical Ambiguity", *Proceedings of the ACL*, Stanford, 6-9 July, 1987.

[Vaughan et al. 86] M. M. Vaughan and D. D. McDonald, "A Model of Revision in Natural Language Generation", *Proceedings of the ACL*, New York, 10-13 June, 1986.

[Wilensky 88] R. Wilensky, D. N. Chin, M. Luria, J. Martin, J. Mayfield, and D. Wu, "The Berkeley UNIX Consultant Project", *Computational Linguistics, Vol. 14, No. 4*, December 1988.

[Wittenburg 86] K. Wittenburg, "A Parser for Portable NL Interfaces Using Graph-Unification-Based Grammars", *Proceedings of AAAI 86*, 1986.