A Complete, Efficient Sentence-Realization Algorithm for Unification Grammar[*]

Robert C. Moore
Microsoft Research
bobmoore@microsoft.com

Abstract

This paper describes an efficient sentence-realization algorithm that is complete for a very general class of unification grammars. Under fairly modest constraints on the grammar, the algorithm is shown to have polynomial time complexity for generation of sentences whose logical form exactly matches the goal logical form. The algorithm can be extended to handle what is arguably the most important subcase of the logical-form equivalence problem, permutation of logical conjunction. With this extension the algorithm is no longer polynomial, but it seems to be about as efficient as the nature of the problem permits.

1 Introduction

We describe an efficient sentence-realization algorithm that is complete for a very general class of unification grammars. Given a goal consisting of a category and a logical form (LF), our algorithm will generate every string classified by the grammar as having the goal category and goal LF. The only constraint required for completeness is a weakened version of Shieber's (1988) semantic monotonicity property. Under fairly modest constraints on the grammar, the algorithm is shown to have polynomial time complexity for strict generation; i.e., not addressing LF equivalence. The exact polynomial depends on details of the grammar, and we discuss the likely order of the polynomial for English.

We extend the algorithm to be complete for the generation of sentences whose LF is equivalent to a goal LF under permutation of n-ary logical conjunction. This is arguably the most important subcase of what has come to be called "the logical-form equivalence problem" (Appelt, 1987; Shieber, 1988, 1993). Allowing permutation of conjunction makes the generation problem inherently worst-case exponential, but our algorithm seems to be about as efficient as one could expect under the circumstances.

2 Grammatical Framework

We assume a grammar formalism having the expressive power of definte clause grammar (Pereira and Shieber, 1987), or equivalently, the syntactically sugared forms used in the Core Language Engine (Alshawi, 1992) or Gemini (Dowding et al., 1993). The following is an example of the sort of grammar rule we assume:

```
s:[stype=decl] -->
     np:[prsn=P,num=N]
     vp:[vtype=tensed,prsn=P,num=N]
```

The notation is that of augmented phrase structure rules, where nonterminals are complex category expressions having the form of a major category symbol followed by a list of feature constraints. Atomic values beginning with uppercase letters are variables; those beginning with lower case letters are constants. Unification constraints are indicated by shared variables. For instance, the sample rule above

[*]Revised version of paper appearing in *Proceedings of the International Natural Language Generation Conference, INLG'02*. Revised 17 December 2002.

would be interpreted to mean that a declarative sentence can be a noun phrase followed by a tensed verb phrase, such that the person and number of the noun phrase are equal to the person and number of the verb phrase, respectively.

To extend the formalism to incorporate semantic specifications assigning an LF to each phrase, we augment the nonterminals with an LF specification, separated by the symbol "/":

```
s:[stype=decl]/VP_sem -->
     np:[prsn=P,num=N]/NP_sem
     vp:[vtype=tensed,prsn=P,num=N,
         sub=NP_sem]/VP_sem
```

The rule now says that the LF of the sentence is the same as the LF of the verb phrase, and the LF of the noun phrase is unified with the `sub` ("subject") feature of the verb phrase. We assume that the verb phrase has an underspecified LF that is completed by incorporating the LF of the noun phrase. Notice this means that phrases not only have a distinguished principal LF (following the "/"), indicating the overall meaning of the phrase, but they can also have LF-valued features. We assume that all LF-valued features are declared as such by the grammar writer.

Lexical items are introduced by rules such as

```
vp:[vtype=tensed,prsn=3,
    num=sg,sub=S]/sleep(S) -->
      sleeps

np:[prsn=3,num=sg]/sue -->
      Sue
```

The first of these rules says that *sleeps* is a third-person, singular, tensed verb phrase, whose LF is of the form `sleep(S)`, where `S` is the value of the `sub` feature. The second rule says that *Sue* is a third-person, singular noun phrase, whose LF is `sue`. These lexical rules illuminate the way our sample phrasal rule works. Suppose we unify the nonterminal expression for *Sue* with the noun phrase daughter of the sentence rule, and unify the nonterminal expression for *sleeps* with the verb phrase daughter of the sentence rule. This will cause the principal LF of the noun phrase, `sue`, to be unified with the `sub` feature of the verb phrase, which will instantiate the principal LF of the verb phrase to `sleep(sue)`, which will in turn become the LF of the entire sentence.

3 The Basic Algorithm

The algorithm we have developed is most similar to that of Shieber (1988). Shieber's generation algorithm is a reworking of Earley's (1970) parsing algorithm, in which all constraints on word selection and word position are removed, and replaced by a filter that discards any item whose principal LF does not unify with some well-formed subexpression of the goal LF. We discuss the differences between our approach and Shieber's after presenting our algorithm.

3.1 Generation Data Structures

Our algorithm is a form of bottom-up chart generation. Like chart parsing, it builds a collection (called a *chart*) of data structures (called *edges*) representing the (partial) application of grammar rules to previously analyzed phrases. We use the following types of edges:

- A *complete edge* $\langle X \rangle$ means that a complete analysis (including words) of the nonterminal X has been generated.
- An *incomplete edge* $\langle A \rightarrow \alpha . X\beta \rangle$ means that the sequence of nonterminals and/or words α has been generated, and if the sequence of nonterminals and/or words $X\beta$ is generated, then nonterminal A will have been generated. (α and/or β can be empty.)
- An incomplete edge is referred to as an *initial edge* if α is empty; e.g., $\langle A \rightarrow .X\beta \rangle$.
- A *completed edge* $\langle A \rightarrow \alpha . \rangle$ means that the sequence of nonterminals and/or words α has been generated, which results in the nonterminal A having been generated.

The main difference between these edge types and those used in chart parsing is that there are no string position indices, since as Shieber noted, they are not needed to constrain generation. Instead, the nonterminals incorporate LF components to indicate what LFs we have generated strings for, and this guides the generation process the way string positions guide parsing. Also note that, since in our formalism semantic content is associated with nonterminals in grammar rules rather than words, we do not need any complete edges in the chart for lexical items (which are used in some, but not all, chart parsing implementations). Instead, we use lexical “scanning” rules that allows us to hypothesize any word wherever we need it, by “moving the dot” in an incomplete edge past any word that occurs immediately to its right.

3.2 Filtering Edges

By eliminating constraints on word selection and word position, any standard bottom-up chart parsing algorithm would turn into a generator, but it would be a completely unconstrained generator producing semantic analyses of all grammatical strings in the language. Following Shieber, then, we constrain the generator to produce only edges whose semantic components (generalized to include LF-valued features) are compatible with well-formed subexpressions of goal LFs. Our filtering process works as follows:

- First enumerate all the well-formed subexpressions of the goal LF components, including a distinguished, semantically null token to account for semantically null words or phrases. Assuming that LFs are tree structured, the number of well-formed subexpressions will be linear in the size of the LF.

- As initial incomplete edges are generated, include an edge only if it is possible to simultaneously unify all *nonvariable* LF components of the edge with well-formed subexpressions of the LF components of the goal.

- As completed edges are generated, instantiate all *nonvariable* LF components of the left-hand side nonterminals to be well-formed subexpressions of the LF components of the goal, in all possible ways. If it is impossible to instantiate a completed edge in this way, it is not generated.

For example, suppose our goal is to generate all strings for the goal

```
s:[]/restlessly(sleep(sue)).
```

The lexical rule

```
vp:[vtype=tensed,prsn=3,
    num=sg,sub=S]/sleep(S) -->
      sleeps
```

will generate the completed edge

```
vp:[vtype=tensed,prsn=3,
    num=sg,sub=sue]/sleep(sue) -->
      sleeps .
```

because the only goal LF subexpression that unifies with `sleep(S)` is `sleep(sue)`.

3.3 Schematic Specification of the Algorithm

We can express our bottom-up chart generation algorithm as an edge derivation schema. In the following schema, A, B, B', and C represent nonterminals; a represents a terminal; X represents either a terminal or nonterminal; α, β, and γ represent (possibly empty) sequences of terminals and nonterminals; $mgu(B, B')$ represents the most general unifier of B and B'; and $I(E, \theta)$ represents the relation that associates with any edge, grammar rule, or nonterminal E every substitution function θ that instantiates every nonvariable LF component of E to be a well-formed subexpression of an LF component of the goal.

1a. $A \rightarrow \alpha B\beta, \exists\theta(I(A \rightarrow \alpha B\beta, \theta)) \vdash$
$\langle A \rightarrow .\alpha B\beta\rangle$

1b. $A \rightarrow \gamma, \neg\exists\alpha B\beta(\gamma = \alpha B\beta), I(A, \theta) \vdash$
$\langle\theta(A \rightarrow \gamma.)\rangle$

2a. $\langle A \rightarrow \alpha.aX\beta\rangle \vdash \langle A \rightarrow \alpha a.X\beta\rangle$

2b. $\langle A \rightarrow \alpha.a\rangle, I(A, \theta) \vdash \langle\theta(A \rightarrow \alpha a.)\rangle$

3a. $\langle A \rightarrow \alpha.BC\beta\rangle, \langle B'\rangle, \sigma = mgu(B, B') \vdash$
$\langle\sigma(A \rightarrow \alpha B.C\beta)\rangle$

3b. $\langle A \rightarrow \alpha.B\rangle, \langle B'\rangle, \sigma = mgu(B, B'), I(A, \theta) \vdash$
$\langle\theta(\sigma(A \rightarrow \alpha B.))\rangle$

4. $\langle A \rightarrow \alpha.\rangle \vdash \langle A\rangle$

There are seven schema rules, the first six of which are paired, with one of each pair producing an incomplete edge and one, a completed edge. Rule 1a produces an incomplete edge for each grammar rule that has at least one nonterminal on the right-hand side, checking to make sure that all nonvariable LF components can be simultaneously instantiated to goal LF subexpressions. Rule 1b immediately produces one or more completed edges for each grammar rule that has no nonterminals on the right-hand side, instantiating all nonvariable LF components to goal LF subexpressions all possible ways. Note that if each grammar rule is indexed by the least-frequently occuring LF atom or functor it contains (including the null LF token, for rules containing no LF atoms or functors), it is not necessary to examine the entire grammar and lexicon to initialize the generation process. It is only necessary to examine phrasal and lexical rules that are indexed by a functor or atom occurring in a goal LF component.

Rules 2a and 2b are lexical scanning rules, hypothesizing a word a wherever we need one. Rule 2a produces incomplete edges, and rule 2b produces completed edges, instantiating nonvariable LF components to goal LF subexpressions. Rules 3a and 3b combine incomplete and complete edges, unifying a complete edge with the nonterminal immediately to the right of the dot in the incomplete edge. Rule 3a produces incomplete edges, and rule 3b produces completed edges, instantiating nonvariable LF components to goal LF subexpressions. Rule 4 produces complete edges from completed edges, by dropping the information specifying the immediate constituents of the left-hand side nonterminal.

Generation is successful if a complete edge is derived for the goal nonterminal, including both the goal category and goal LF. All complete analyses of the goal nonterminal can be recovered by tracing back through the completed edges. Since each analysis tree has the corresponding word sequence as its leaf nodes, the generated strings can be extracted in this way.

Although string positions play no role in the generation algorithm, generation of each constituent nevertheless proceeds by matching daughters left to right, which makes it easy to keep track of the order of constituents for extracting the generated strings. The algorithm can be made more efficient in practice (though theoretical worst-case complexity may be unaffected), by matching first on the daughter constituting the *semantic head* as defined by Shieber et al. (1990). This is because the category and LF of the semantic head usually constrain the LFs of the other daughters, but not vice versa. Such a semantic-head-based processing order requires some modification to the representation of edges, so that the order of processing and constituent order are represented separately.

3.4 Comparison to Shieber's Algorithm

Our algorithm differs from Shieber's in two principal ways:

- Ours is based on bottom-up chart parsing rather than Earley's algorithm.
- Ours filters chart edges differently than Shieber's does.

We drop Earley-style prediction because it frequently fails to pass along any semantic constraints. For instance, if our goal is to generate a sentence with a particular LF, and we apply Earley prediction to the sentence rule in Section 2, we will predict a noun phrase, but with no constraint on its LF, because there is no direct connection in the sentence rule between the LF of the sentence and the LF of the noun phrase. That connection is made only later, when the verb phrase is realized.

The more important difference between Shieber's algorithm and ours is in the filtering of edges, however. At all stages of processing, Shieber checks the principal LF of each edge to make sure it is unifiable with some goal LF subexpression, but he never instantiates edges in this process. We, on the other hand, instantiate every nonvariable LF component of the left-hand side nonterminal of every completed edge (and therefore every complete edge).

This has two significant advantages. First, it reduces the number of possible distinct completed and complete edges, since for every possible instantiation of an LF component of a completed or complete edge, Shieber will also allow all possible generalizations of that instantiation. Second, it enables us to reduce the number of LF expressions we have to examine to ensure compatibility with goal LF components. Since an LF expression cannot change once it is fully instantiated, if we tag each LF expression that has been instantiated to be a goal LF subexpression, we never have to check that LF expression again. If LF expressions remain partly instantiated, as in Shieber's algorithm, they must be rechecked as they are percolated from edge to edge, since they might become further instantiated in ways incompatible with any goal LF subexpression.

4 Semantic Monotonicity and Completeness

Shieber realized that filtering out chart edges whose LF was not unifiable with a subexpression of the goal LF would not permit complete generation for an arbitrary grammar, so he proposed the following *semantic monontonicity* condition as a constraint guaranteeing completeness:

> "A grammar is semantically monotonic if, for every phrase admitted by the grammar, the semantic structure of each immediate subphrase subsumes some portion of the semantic structure of the entire phrase." (Shieber, 1988, p. 617)

In other words, the LF of every phrase has to incorporate the LFs of all its daughters. Some constraint of this sort seems reasonable, but this definition of semantic monotonicity is stronger than it needs to be in two ways. First, it makes no allowance for semantically null constituents. In the sentence *Oh my goodness, I didn't realize you were here*, we might wish to regard the exclamation *Oh my goodness* as semantically null. We therefore designate a distinguished token representing the null meaning, and stipulate that it constitutes a well-formed subexpression (a "portion" to use Shieber's term) of every LF expression.[1]

The second problem with Shieber's definition is that it does not take account of LF-valued features, in addition to the principal LF expressing the overall meaning of a phrase. These LF-valued features could be used to carry semantic information that is not used locally—for example, a the meaning of a modifier extraposed from some other constituent—which is passed up the analysis tree, but does not form part of the principal LF of an immediate parent. In this case Shieber's semantic monotonicity test would not be satisfied. We therefore revise Shieber's definition as follows:

> A grammar is semantically monotonic if, for every phrase admitted by the grammar, each semantic component (principal LF, or LF-valued feature) of each immediate subphrase subsumes some well-formed subexpression (including the null subexpression) of some semantic component of the entire phrase.

Since the only way for a phrase to pass its semantic contribution up to the goal is through the phrase's immediate parent, this condition seems to be the weakest constraint one could ask for that does not allow grammars to simply throw semantically significant information away.

Viewed abstractly, our generation algorithm is a free bottom-up generator, with the addition of an instantiation process that forces each nonvariable LF component of a complete edge to be a goal LF subexpression. The only derivations this eliminates, however, are those that violate our revised definition of semantic monontonicity. Our algorithm is therefore complete for any semantically monotonic grammar.

5 Computational Complexity of the Algorithm

How efficient is our algorithm? If we choose a dynamic programming implementation of the abstract algorithm, then two or more identical edges will never be added to the chart. With this assumption, we can analyze the complexity of our algorithm as a function of the total size of the goal LF components, n, as being of the order of the sum of two terms:

$$Gn^q + GC^k n^r$$

G represents the number of grammar rules and q represents the maximum number of partially or fully instantiated LF components in any grammar rule. Gn^q is therefore a bound on the number of steps (counting a unification as a single step) needed to check these grammar rules against all goal LF subexpressions according to generation schema rules 1a and 1b.

C represents the number of possible complete edges, k represents the maximum number of nonterminals on the right-hand side of a grammar rule, and r represents the maximum number of instantiated LF components in the left-hand side of a completed edge that are not inherited fully instantiated from one of the complete edges used in the derivation the completed edge. So, each of at most G initial edges

[1]Note the obvious analogy with the null string being regarded as a substring of every string.

can be combined with at most C^k combinations of complete edges to produce a completed edge, which can be further instantiated in at most n^r steps (assuming that already fully-instantiated LF components are tagged as such), which comes to GC^kn^r. The other two types of generation steps, scanning lexical items and producing complete edges from completed edges, at worst multiply these terms by small constants, and thus do not increase the overall complexity of the algorithm.

The first term of this complexity bound is clearly polynomial, but to complete the analysis of the second term, we need to further analyze C, the number of possible complete edges. With an unrestricted unification grammar, there may be no bound at all on C; however, it is usually possible to put a bound on C in practice. Our instantiation strategy ensures that each LF component of a complete edge has at most order n possible values. Suppose that there are at most s maximal sets of *strongly-dependent* LF components in any nonterminal.[2] Suppose further that for a given combination of LF components of a nonterminal, the number of combinations of non-LF-valued features (i.e., syntactic categories) is bounded by a polynomial of order t. In this case, $C = n^{st}$, and the whole second term becomes Gn^{stk+r}, so the complexity of the entire algorithm is polynomial.

Except for a handful of languages having cross-serial dependencies, most natural languages seem to be describable using grammars that have an absolute upper bound on the number of distinct syntactic categories, which would limit t to be 1. (A sufficient condition for this, which can be easily enforced in practice, is to define all non-LF-valued features to have only a finite set of values.) Moreover, in this case, the number of independent LF components in a given nonterminal *must* have a fixed upper bound, since the finite set of distinct syntactic categories provides only a fixed set of LF-valued "slots" for LF components. In this case, then, the bound on the second term of our complexity formula simplifies to Gn^{sk+r}. If the grammar is binary branching, this becomes Gn^{2s+r}. Furthermore, grammars are often written to be semantically lexicalized, so that every nonvariable LF component of a nonlexical completed edge is inherited fully instantiated from one of the complete edges used in its derivation. In this case $r = 0$, and the second term is reduced to Gn^{2s}.

The value of s in this expression depends very much on the details of the grammar. We conjecture that for English, the best we can do may be $s = 3$. The nonterminals seeming to require the largest number of maximal strongly-dependent sets of LF components we are aware of are those with two syntactic gaps, as in the infinitive phrase in the well-known example *Which violin are these sonatas easy to play _ on _?* It seems possible to elaborate such phrases so that as the size of the overall phrase (and its LF) increases, the number of choices for the two gap fillers that are not strongly dependent on the principal LF also increases linearly. Since no construction of English is known to require more than two syntactic gaps, we conjecture $s = 3$, for the principal LF and the LFs of the two gap fillers. Plugging $s = 3$ into n^{2s}, would give us n^6 generation steps for this term, which we presume dominates the other term of the complexity formula.

The possible bounds we have been discussing up to now are in terms of "generation steps", that count a unification as part of a single step. Unification can be performed in linear time, and under the assumption that the number of syntactic categories is bounded, this would add at most another factor of n to the overall time complexity of our algorithm. However, by tagging all the well-formed subexpressions of the goal LF,[3] we can implement a restricted form of unification that stops when it reaches a comparison of two well-formed LF subexpressions. In the case where only a bounded number of syntactically distinct nonterminals can arise, this results in all unifications being depth-bounded, which can (almost) be performed in constant time. The "almost" comes from the fact that we require on the order of $log_2(n)$ symbols to tag n well-formed LF subexpressions, and in principle comparing those requires on the order of $log_2(n)$ time. However, on any computer that has 32-bit equality comparison as

[2]A set of LF components is strongly dependent, if there is a member of the set such that selecting a value for that member puts fixed bounds on the number of possible values for the other members of the set. For example, choosing a particular instantiation of the principal LF of a verb phrase might leave at most three choices for the value of the subject feature (the subject, direct object, and indirect object of the active form of the verb).

[3]These can be the same tags mentioned in Section 3.4.

a single operation, we can handle such unifications in bounded time with up to 4 billion such symbols.

6 Handling Logical-Form Equivalence

The bottom-up chart generation algorithm presented above is both complete and efficient in theory, but it is subject to what has been called "the logical-form equivalence problem" (Appelt, 1987; Shieber, 1988, 1993). The generator, as described, takes the association of strings and LFs specified by the grammar as being exact, with no accomodation for the fact that some differences in LF notation might be semantically insignificant. For example, a grammar might give *Sue sees Mary* the LF

`[see(E),agt(E,sue),pat(E,mary)],`

but give *Mary is seen by Sue* the LF

`[see(E),pat(E,mary),agt(E,sue)].`

If we interpret these expressions as simple conjunctions of three atomic formulas, the order of the conjuncts should be irrelevant. Our generator, however, would generate only *Sue sees Mary* from

`[see(E),agt(E,sue),pat(E,mary)],`

and only *Mary is seen by Sue* from

`[see(E),pat(E,mary),agt(E,sue)].`

because nothing indicates to the generator that the order of the items in these implicit conjunctions is semantically irrelevant.

Over the past decade a number of approaches to generation have been developed that address this form of logical equivalence (Brew, 1992; Kay, 1996; Carrol et al., 1999). All of these approaches, however, place restrictions on the organization of the grammar and/or the logical form notation. We address this problem by providing a single special notation in our LF language to indicate when elements of an LF subexpression may be permuted without changing the meaning of the LF. We use square-bracketed lists for this purpose; e.g., `[a,b,c]`, `[a,c,b]`, `[b,a,c]`, etc. will all be treated equivalently. We modify our algorithm to be complete with respect to the new interpretation of our LF notation.

The principal change we make to our generator is to modify the unification algorithm so that lists consisting of the same elements in different orders will unify. We also need to add to our enumeration of well-formed subexpressions of goal LFs all subsets of each list subexpression, maintaining the order of the original list. Thus if `[a,b,c]` is a subexpression of the goal LF, then we also need to treat `[a,b]` and `[a,c]` as well-formed subexpressions (as well as all possible tail segments of the list, which were already included), but not `[b,a]` or `[c,a]`.

Since every complete edge has all of its nonvariable LF components unified with goal LF subexpressions, which are always fully instantiated, we can choose to represent the result of these unifications by the goal LF subexpressions involved. This means these in effect become canonical forms for equivalence classes of the LF expressions we need to deal with. Thus, for complete edges, we not only have no duplication of identical edges in the chart; we also have no duplication of equivalent edges in the chart.

No other changes to our framework or algorithm are required. In particular, there is no requirement that the grammar be semantically lexicalized, and no limitations are placed on the LF notation as to what other constructs can be used besides (square-bracketed) lists. Some care in the design of the LF notation can enhance efficiency, however. We treat all subsets of lists that are goal LF subexpressions as goal LF subexpressions to allow for the possibility that the elements of the list may occur in all possible orders. If this is not the case, then efficiency of generation can be improved by modifying the notation to reflect this fact.

For example if we let

`[see(E),pat(E,mary),agt(E,sue)]`

be a possible LF for *Sue sees Mary*, then we must treat `[see(E),pat(E,mary)]` as a possible well-formed subexpression, in case `agt(E,sue)` is later added to the front of the list to produce the LF

`[agt(E,sue),see(E),pat(E,mary)],`

which would be equivalent. In the grammar-writing style we have adopted, however, the LF element corresponding to the lexical head of a phrase (in this case *sees* always comes first in the list, no matter what order the complements or adjuncts are in. If we reflect this in the notation, by letting the LF for *Sue sees Mary* be

`see(E,[pat(E,mary),agt(E,sue)]),`

or some other notation that fixes the position of the head, then we can avoid generating a string for `[see(E),pat(E,mary)]` that spuriously omits required complements and adjuncts.

The fact that we have to admit as a well-formed LF subexpression every subset of each list that is a well-formed LF subexpression means that, in the worst case, every factor of n in the analysis of the number of generation steps in the original algorithm would be replaced by a factor of 2^n in the extended algorithm.[4] However, when list permutation is permitted, it is easy to demonstrate grammars that require exponentially many chart edges to represent all analyses of a given LF; so it is not clear that any significantly more efficient algorithm is possible in principle.

With the addition of list permutation, our method turns out to resemble the approaches of Kay (1996) and Carrol et al. (1999) in a many respects. The greatest advantage of our approach, however, is its almost complete freedom of choice as to LF notation, whereas these other approaches require "flat" LFs. While our approach does require some care in choice of notation for maximum efficiency, it remains less restrictive than the others, and the algorithm still functions correctly with no restrictions on LF notation.

7 Conclusions

We have implemented our algorithm in both a strict and list-permuting version, and verified that it performs as expected on a small test grammar. It is complete for a wider class of grammars than Shieber's algorithm, and seems to be more efficient. Its generalization to handle a subset of the LF-equivalence problem requires only minor changes to the approach, places no restrictions on the form of the grammar or LF notation, and seems about as efficient as the nature of the problem permits.

References

Hiyan Alshawi, ed. 1992. *The Core Language Engine*. The MIT Press, Cambridge, Massachusetts.

Douglas Appelt. 1987. Bidirectional grammars and the design of natural language generation systems. In *Theoretical Issues in Natural Language Processing—3*, pages 185–191, New Mexico State University, Las Cruces, New Mexico.

Chris Brew. 1992. Letting the cat out of the bag: generation for shake-and-bake MT. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 610–616, Nantes, France.

John Carroll, Ann Copestake, Dan Flickinger and Victor Poznański. 1999. An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of the 7th European Workshop on Natural Language Generation (EWNLG'99)*, pages 86–95, Toulouse, France.

John Dowding, J. Mark Gawron, Douglas Appelt, John Bear, Lynn Cherny, Robert Moore, and Douglas Moran. 1993. Gemini: a natural language system for spoken-language understanding. In *Proceedings of the 31st Annual Meeting of the Association for Computational Linguistics*, pages 54–61, Columbus, Ohio.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Martin Kay. 1996. Chart generation. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 200–204, Santa Cruz, California.

Fernando Pereira and Stuart Shieber. 1987. *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information, Stanford University, Stanford, California.

Stuart Shieber. 1988. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 614–619, Budapest, Hungary.

Stuart Shieber, Gertjan van Nord, Fernando Pereira, and Robert Moore. 1990. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42.

[4]Unification also becomes at least quadratic.

Stuart Shieber. 1993. The problem of logical-form equivalence. *Computational Linguistics*, 19(1):179–190.