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 be polynomial time 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 be polynomial time for strict generation; i.e., not addressing LF equivalence. The exact polynomial depends on details of the form 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.

*Revised version of paper submitted to International Natural Language Generation Conference, INLG'02. Do not quote without permission. Revised 10 March 2002.

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:[person=P,num=N]
    vp:[vtype=tensed,person=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 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:[person=P,num=N]/NP_sem
    vp:[vtype=tensed,person=P,num=N,subj=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 `subj` 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 by the grammar writer.

Lexical items are introduced by rules such as

```
v:[vtype=tensed,person=3,num=sg,subj=S]/sleep(S) --> sleeps
```

This rule says that *sleeps* is a third-person, singular, tensed verb, whose LF is of the form `sleep(S)`, where `S` is the value of the `subj` feature. This lexical rule illuminates the way our sample phrasal rule works. If the `subj` feature and LF of the verb and the verb phrase are constrained to have the same respective values, and we assign the `subj` feature of the verb phrase a specific value, say, `john`, then the LF of the verb will become `sleeps(john)`, which will in turn become the LF of the verb phrase and the entire sentence.

3 Semantic Monotonicity

The algorithm presented here 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 phrases whose LF does not subsume some well-formed subexpression of the goal LF. Shieber realized that this filter 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)

Semantic monotonicity as defined by Shieber 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.

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 can be used to carry semantic information that is not used locally—for example, a modifier extraposed from some other constituent—which would fail Shieber's semantic monotonicity test. 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. In our generation algorithm, we apply a filter appropirate to this revised definition of semantic monotonicity.

4 The Basic Algorithm

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

- It is based on bottom-up chart parsing rather than Earley's algorithm.
- It filters chart edges differently.
- The filter is applied only at the time the chart is initialized.

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 the empty sequence; e.g., $\langle A \rightarrow .\beta \rangle$.
- 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 add a generation "scanning" rule 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.

With only these changes, any standard bottom-up chart parser 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 (including the semantically null subexpression). We do this, however, in a way that requires constraining only initial edges. Shieber's filter only checks that an LF expression subsumes some goal LF subexpression, without changing the expression being tested. Our algorithm instantiates LF expressions to be identical to goal LF subexpressions. In particular, we initialize the chart by instantiating all *nonvariable* LF components of the mother nonterminal of an initial edge all possible ways with well-formed subexpressions of the LF components of the goal. Any grammar rule having an LF component that does not unify with some goal LF subexpression fails to generate any initial edges.

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

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

The lexical rule

```
v:[vtype=tensed,person=3,num=sg,subj=S]/sleep(S) --> sleeps
```

will produce the initial edge

```
v:[vtype=tensed,person=3,num=sg,subj=john]/sleep(john) --> . sleeps
```

by instantiating the LF `sleep(S)` to match the goal LF subexpression `sleep(john)`. Notice that the `subj` feature is also instantiated to have the value `John` as a side effect. On the other hand, the phrasal rule

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

will be transformed into an initial edge without instantiation, because the only LF component of the mother nonterminal has a completely uninstantiated variable as its value. We let $I(A, \theta)$ be a relation that associates with a nonterminal A every substitution function θ that satisfies these constraints.

We now define our version of bottom-up chart generation by the following schema:

1. $A \to \alpha, I(A, \theta) \vdash \langle \theta(A) \to .\theta(\alpha) \rangle$
2. $\langle A \to \alpha.a\beta \rangle \vdash \langle A \to \alpha a.\beta \rangle$
3. $\langle A \to \alpha. \rangle \vdash \langle A \rangle$
4. $\langle A \to \alpha.X\beta \rangle, \langle X' \rangle, \theta = mgu(X, X') \vdash \langle \theta(A) \to \theta(\alpha)\theta(X).\theta(\beta) \rangle$

The first generation rule initializes the chart with initial edges instantiated as we have described. The second rule allows us to hypothesize a word a wherever we need one. The third and fourth rules are unchanged from what one might see in a chart parsing schema, except for the removal of the sentence position indices. 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.

Although our algorithm directly constrains only the mother nonterminals of initial edges so that the value of every LF component is either a variable or a goal LF subexpression, it can be shown by induction that every edge added to the chart has this property. Thus, like Shieber's, our algorithm never adds to the chart any edge that cannot participate semantically in deriving the goal LF. Moreover, we do not incur the cost of applying the filter to every generated edge, or the cost of generating edges that fail to pass the filter.

5 Completeness and Efficiency

Our instantiation process for initial edges rules out only those instances that include some LF component that cannot unify with any goal LF subexpression. Otherwise, our abstract generation algorithm implements free bottom-up generation. Therefore, the algorithm will be complete unless the grammar licenses some derivation that requires some intermediate nonterminal to have an LF component whose value will not unify with any goal LF subexpression (including the semantically null subexpression). For grammars that rule this out by following our revised semantic monotonicity condition, the algorithm will be complete.

How efficient is our algorithm? If we choose a dynamic programming implementation of the abstract algorithm, then no edge will be added to the chart more than once. Under this assumption we can show, by a derivation that space does not permit us to repeat here, that the total number of generation steps to construct a complete chart is bounded by a function of order $n^{L(n)}(S(n^{L(n)}))^k$, where n is the total size of the goal LF components; k is the maximum branching factor of the grammar; L is a function that determines, given n, the maximum number of independent LF components a nonterminal can have; and S is a function that determines the total number of syntactic variants there can be of a nonterminal with a given combination of LF components.

In general L and S are not guaranteed to exist; there may be no bounds on the quantities they purport to bound. In this case, our algorithm cannot be guaranteed to terminate, although it can still be complete in the sense of eventually generating every possible analysis. It seems reasonable to assume that L and S exist for grammars of natural languages, however. Psychological plausibility suggests that understanding a phrase should not require keeping track of an unbounded number of unrelated semantic elements, so L should exist; and assuming the existence of S amounts to nothing more than requiring that there not be infinitely many syntactic ways of expressing the same meaning.

The actual computational complexity of our algorithm will depend on what functions L and S are, and on the complexity of the individual generation steps. If L is bounded by a constant l and S is a polynomial of order s, then the complexity in terms of generation steps is $n^l((n^l)^s)^k = n^{(sk+1)l}$, which is polynomial in the size of the input, n.

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 s to be 1. (A sufficient condition for this, which can be easily enforced in practice, is to define all syntactic 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 cateogories provides only a fixed set of LF-valued "slots" for LF components. In this case, then, the bound on the complexity of the algorithm simplifies to $n^{(k+1)l}$. If, in addition, the grammar is limited to binary branching, then $k = 2$ and the number of generation steps is no more than order n^{3l}.

The value of l in this expression depends very much on the details of the grammar. We conjecture that for English, the upper bound may be $l = 3$. The nonterminals requiring the largest number of independent 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 independent choices for the two gap fillers also increases linearly. Since no construction of English is known to require more than two syntactic gaps, we conjecture $l = 3$, for the principal LF and the LFs of the two gap fillers. Plugging $l = 3$ into n^{3l}, would give us n^9 generation steps. This bound assumes that any nonterminal can potentially combine with any other nonterminal in a binary-branching grammar, which actually seems to be a looser constraint than necessary. We are continuing to investigate additional plausible constraints on English that might give us a bound lower than n^9.

The possible bounds we have been discussing up to now are in terms of "generation steps", by which we mean the application of one of our generation schema rules. Executing one generation step requires scanning the edges involved and performing at most one unification. Using an efficient, general unification algorithm, this can be carried out in time linear in n, which would add another factor of n to the overall time complexity of our algorithm. However, by tagging all the well-formed subexpressions of the goal LF, 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 a single operation, we can handle such unifications in bounded time with up to 4 billion such symbols. Thus in practical terms, the unifications can be done in bounded time.

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 "John sees Mary" the LF

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

but give "Mary is seen by John" the LF

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

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 "John sees Mary" from

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

and only "Mary is seen by John" from

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

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

While it may also be desirable for a generator to handle other types of LF equivalence in addition to reordering of conjuncts, over the past decade a number of approaches to generation have been developed that focus on this specific problem (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]`. It is a property of our algorithm that every LF unification involves a descendant (in terms of iterated unifications) of a goal LF subexpression, all of which are already fully instantiated. We therefore choose to represent the result of all LF unifications by goal LF subexpressions involved, which means that these, in effect, become the canonical forms for every equivalence class of LF expressions that we need to deal with. Thus, 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, no assumptions are made about the grammar being "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,john)].
```

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

```
[agt(E,john),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 compliments or adjuncts are in. If we reflect this in the notation, by letting the LF for "John sees Mary" be

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

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 compliments and adjuncts.[1]

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 a complexity bound on the original algorithm would be replaced by a factor of 2^n in the extended algorithm. 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, out 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.

[1]An alternative approach to this problem is to use a list-permuting, chart-based version of semantic-head-driven generation (Shieber et al., 1990). The goal-directed nature of this approach would avoid generating strings for spuriously incomplete subsets of lists, but it imposes other restrictions on the form of the grammar that can sometimes be difficult to accomodate.

References

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

D. 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.

C. 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.

J. Carroll, A. Copestake, D. Flickinger and V. Poznanski. 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.

J. Dowding, J. M. Gawron, D. Appelt, J. Bear, L. Cherny, R. Moore, and D. 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.

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

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

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

S. Shieber. 1988. A uniform architecture for parsing and generation. In *Proceedings of the 14th International Conference on Computational Linguistics*, pages 614–619, Budapest, Hungary.

S. Shieber, G. van Nord, F. Pereira, and R. Moore. 1990. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42.

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