EXPLOITING F-STRUCTURE INPUT FOR SENTENCE CONDENSATION

Richard Crouch, Tracy Holloway King, John T. Maxwell III,
Stefan Riezler, and Annie Zaenen
Palo Alto Research Center

Proceedings of the LFG04 Conference
University of Canterbury
Miriam Butt and Tracy Holloway King (Editors)

2004

CSLI Publications
http://csli-publications.stanford.edu/

Abstract

In this paper, we describe the types of sentence condensation rules used in the sentence condensation of Riezler et al. 2003 in detail. We show how the distinctions made in LFG f-structures as to grammatical functions and features make it possible to state simple but accurate rules to create smaller, well-formed f-structures from which the condensed sentence can be generated.

1 Introduction

The aim of sentence condensation is to keep only the essential elements of a sentence and the relations among them and to produce a grammatical, condensed output, which can then be used in applications such as text summarization. Keeping the essential elements could conceivably be achieved by selecting key terms; however, keeping the relations and producing grammatical output requires a structure-based approach. An example of sentence condensation is shown in (1).

(1) The UNIX operating system, with implementations from Apples to Crays, appears to have the advantage. $\Longrightarrow$ UNIX appears to have the advantage.

In this paper, we show how the LFG formalism, in particular the f(unctional)-structure, provides the necessary format for manipulating sentences in a condensation system. In particular, LFG f-structures provide the necessary distinctions to alter the structure of the sentence and then allow for the generation of grammatically well-formed output strings. For example, LFG allows the easy statement of rules which delete adjuncts (2a), passivize active sentences (2b), delete conjuncts from coordinate structures (2c), and decleft sentences (2d).

(2) a. They left quickly. $\Longrightarrow$ They left.

b. They shattered the glass. $\Longrightarrow$ The glass was shattered.

c. John and Mary left. $\Longrightarrow$ John left.

d. It is Mary who left. $\Longrightarrow$ Mary left.

The paper is organized as follows. Section 2 describes the basic system: the parser, transfer component, and generator. Section 3 comprises the bulk of the paper. It describes the sentence condensation rules themselves and demonstrates how these apply to the LFG f-structure analyses, focussing on adjunct deletion, passivization/activization, and factive verb deletion. Finally, section 4 provides a brief conclusion.

2 The Basic System

Our basic sentence condensation system (Riezler et al. 2003) was developed on and runs in the XLE development environment (Maxwell and Kaplan 1993). In this system, to produce a condensed version of a sentence, the sentence is first parsed using a broad-coverage LFG grammar for English. The parser produces a set of f-structures for an ambiguous sentence in a packed format. It presents these to the transfer component in a single packed data structure that represents in one place the substructures shared by several different interpretations. The transfer component operates on these packed representations and modifies the parser output to produce reduced f-structures.

The reduced f-structures are then filtered by the generator to determine syntactic well-formedness. A stochastic disambiguator using a maximum entropy model was trained on parsed and manually disambiguated f-structures for pairs of sentences and their condensations. Using the disambiguator, the string generated from the most probable reduced f-structure produced by the transfer system is chosen. In contrast to many approaches to sentence condensation (e.g., Knight and Marcu 2000, Witbrock and Mittal 1999), our system guarantees the grammaticality of generated strings through the use of the XLE constraint-based generator for LFG which uses a slightly tighter version of the grammar than is used by the parser.

This section describes the XLE parser, transfer component, and generator. XLE consists of cutting-edge algorithms for parsing and generating LFG grammars along with a rich graphical user interface for writing and debugging such grammars.

2.1 The Parser

One of the main goals of XLE is to efficiently parse and generate with LFG grammars. This is difficult because the LFG formalism, like most unification-based grammar formalisms, is NP complete. This means that in the worst case the time that it takes to parse or generate with an LFG grammar can be exponential to the length of the input. However, natural languages are mostly context-free equivalent, and one should be able to parse them in mostly cubic time. XLE is designed to automatically take advantage of context-freeness in the grammar of a natural language so that it typically parses in cubic time and generates in linear time when the input is fully specified. This allows grammar writers to write grammars in the expressive LFG formalism without necessarily sacrificing performance.

There are three key ideas that XLE uses to make its parser efficient. The first idea is to pay careful attention to the interface between the phrasal and functional constraints (Maxwell and Kaplan 1993). In particular, XLE processes all of the phrasal constraints first using a chart, and then uses the results to decide which functional constraints to process. This is more efficient than interleaving phrasal and functional constraints because the phrasal constraints can be processed in cubic time, whereas the functional constraints may take exponential time. Thus, the phrasal constraints make a good polynomial filter for the functional constraints.

The second key idea is to use contexted unification to merge multiple feature structures together into a single, packed feature structure (Maxwell and Kaplan 1991). For instance, (3) is a contexted feature structure for the sentence *They saw the girl with the telescope*. In this sentence, *saw* is ambiguous between the present tense of *saw* (labeled `a:1`) and the past tense of *see* (`a:2`). Also, *with the telescope* is either an adjunct of *saw* (`b:1`) or *the girl* (`b:2`). The contexts `a:1` and `a:2` are mutually exclusive. Similarly for `b:1` and `b:2`. Furthermore, `a:1` and `a:2` are independent from `b:1` and `b:2`. So this feature structure represents four different solutions. XLE uses a contexted feature structure like this for each edge in a chart to represent all of the possible feature structures that the edge can have.

(3)
$$\begin{bmatrix}
\text{PRED} & \begin{bmatrix} \texttt{a:1} & \text{'saw<SUBJ,OBJ>'} \\ \texttt{a:2} & \text{'see<SUBJ,OBJ>'} \end{bmatrix} \\
\text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'pro'} \\ \text{PRON-FORM} & \text{they} \end{bmatrix} \\
\text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'girl'} \\ \text{SPEC} & \begin{bmatrix} \text{PRED} & \text{'the'} \end{bmatrix} \\ \text{ADJUNCT} & \texttt{b:2} \left\{ [\ 1\text{:with}\ ] \right\} \end{bmatrix} \\
\text{ADJUNCT} & \texttt{b:1} \left\{ \begin{bmatrix} \text{PRED} & \text{'with<OBJ>'} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'telescope'} \\ \text{SPEC} & \begin{bmatrix} \text{PRED} & \text{'a'} \end{bmatrix} \end{bmatrix} \text{:1} \end{bmatrix} \right\} \\
\text{TENSE} & \begin{bmatrix} \texttt{a:1} & \text{present} \\ \texttt{a:2} & \text{past} \end{bmatrix}
\end{bmatrix}$$

The third key idea is to use lazy contexted copying during unification (Maxwell and Kaplan 1996). Lazy contexted unification only copies up as much of the two daughter feature structures of a subtree as is needed to determine whether the feature structures are unifiable. If the interactions between the daughter feature structures are bounded, then XLE will do a bounded amount of work per subtree. Since there are at most a cubic number of subtrees in the chart, then XLE can parse sentences in cubic time when the interactions between daughter feature structures are bounded.

For more details on the XLE parser, see Maxwell and Kaplan 1996.

2.2 The Transfer System

The transfer system is a general purpose packed rewriting system. In the sentence condensation application described in this paper, the transfer system takes a packed f-structure produced by the parser, converts it a to a (multi)set of packed input facts, transfers this to a set of packed output facts, and converts this into a packed output f-structure. The generator then produces the condensed strings from the transfer output.[1]

The transfer rules for sentence condensation consist of an ordered set of rules that rewrite one f-structure into another. Structures are broken down into flat lists of facts, and rules may add, delete, or change individual facts. Rules may be optional or obligatory. In the case of optional rules, transfer of a single input structure may lead to multiple alternate output structures.

Packed transfer rewriting is significant, since an ambiguous sentence gives rise to more than one structure and it is possible for the number of structures to be in the thousands. However, the set of structures for a given sentence is packed into a single representation making it possible for common parts to appear only once (cf. (3) in the previous section). Thanks to the fact that the structures are packed, common parts of alternative structures can often be operated on by the transfer system as a unit, the results being reflected in each of the alternatives.

The transfer system operates on a source f-structure, represented as a set of (transfer) facts, to transform it, little by little, into a target structure. This operation is controlled by a transfer grammar consisting of a list of rules. The order of these rules is important because each rule has the potential of

[1]While this paper focuses on f-structure to f-structure transfer, the transfer system is not restricted to applications involving f-structure input or output, but instead is a general purpose rewrite system.

changing the situation that following rules will encounter (see section 3.1.4). In this respect, the rules are like phonological rules of the Chomsky/Halle variety. In particular, rules can prevent later rules from applying by removing material that they would otherwise have applied to (bleeding) or they can enable the application of later rules by introducing material that they need (feeding). In this respect, this is different from other systems that have been proposed for transfer that build new structures based on observation, but not modification, of existing ones. In this system, as the process continues, the initial source f-structure takes on more and more of the properties of a target f-structure. Source facts that no rule in the sequence applies to simply become part of the target structure. As such, the transfer process never fails. Even if no rules apply, the output would simply be identical to the input. This is crucial for sentence condensation since many shorter sentences will remain unchanged by the rules.

2.2.1 Transfer Facts

The first thing to consider is the nature of transfer facts, and then how f-structures are converted to transfer facts. Transfer facts typically are encoded as a predicate, an opening parenthesis, a comma separated list of arguments,[2] and a closing parenthesis, as in (4). Since sentence condensation applies to f-structures, most facts involve two arguments.

(4) predicate(argument1, argument2)

Predicates are atomic symbols, while arguments can be either atomic or non-atomic (i.e. embedded predicates).

Consider how the f-structure in (5) for the sentence *Mary sleeps.* can be represented as a set of transfer facts, shown in (6).

(5)
$$\begin{bmatrix} \text{PRED} & \text{'sleep<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \text{PERS} & 3 \\ \text{NUM} & \text{sg} \end{bmatrix}_1 \\ \text{TNS-ASP} & \begin{bmatrix} \text{TENSE} & \text{present} \\ \text{MOOD} & \text{indicative} \end{bmatrix}_2 \\ \text{STMT-TYPE} & \text{declarative} \end{bmatrix}_0$$

[2] It is also possible to have atomic facts with no arguments: predicate.

(6)
```
PRED(var(0),sleep)
SUBJ(var(0),var(1))
STMT-TYPE(var(0),declarative)
TNS-ASP(var(0),var(2))
arg(var(0),1,var(1))
lex_id(var(0),3)
```
the outermost f-structure

```
PRED(var(1),Mary)
lex_id(var(1),1)
NUM(var(1),sg)
PERS(var(1),3)
```
the SUBJ f-structure

```
MOOD(var(2),indicative)
TENSE(var(2),pres)
```
the TNS-ASP f-structure

To see how these facts correspond to the f-structure, it is first necessary to understand the convention lying behind the use of `var(n)` arguments. These are to be interpreted as standing for f-structure nodes. Thus the outermost node labeled 0 in the f-structure is represented by `var(0)`, and the value of the SUBJ attribute labeled as 1 is represented by `var(1)`. The same holds for f-structures without PREDs, as seen for the value of the TNS-ASP attribute. This is assigned the index `var(2)` in the transfer facts.

Looking at the facts with `var(0)` as their first argument, most of them correspond directly to attribute value pairs in the f-structure. The fact that f-structure 1 is the value of the SUBJ attribute of f-structure 0 is represented as `SUBJ(var(0), var(1))`. The fact that the value of the STMT-TYPE attribute of f-structure 0 is `declarative` is represented as `STMT-TYPE(var(0), declarative)`. The fact that the value of the TNS-ASP attribute of f-structure 0 is a complex structure is represented as `TNS-ASP(var(0), var(2))`.

The representation of the semantic forms, i.e., the PREDs, is described below in section 3.1. Basically, PREDs are decomposed into the predicate name itself, e.g. *sleep*, the arguments, e.g., SUBJ, and an identifier, e.g., `lex_id`. Each of these can be manipulated separately by the transfer rules, although in sentence condensation this is rarely necessary.

A final point to be made about the representation of f-structures involves sets. Consider the f-structure in (7) for the noun phrase *big black dogs*.

(7)
$$\begin{bmatrix} \text{PRED} & \text{'dog'} \\ \text{NUM} & \text{pl} \\ \text{ADJUNCT} & \left\{ \begin{bmatrix} \text{PRED} & \text{'big'} \end{bmatrix}_2 \\ \begin{bmatrix} \text{PRED} & \text{'black'} \end{bmatrix}_3 \right\}_1 \end{bmatrix}_0$$

The ADJUNCT of the f-structure is a structure enclosed in curly brackets, the standard representation for sets. In the transfer system, sets are represented by the `in_set` predicate. For the structure in (7), it would be as in (8).

(8)
```
ADJUNCT(var(0), var(1))
in_set(var(2), var(1))
in_set(var(3), var(1))
```

The value of 0's ADJUNCT attribute is a set value, that we have indexed as `var(1)`. There are two items in the set, namely `var(2)` and `var(3)`. The use of the `in_set` predicate is demonstrated in section 3 for the deletion of adjuncts in sentence condensation.

2.2.2 Transfer Rules

Having seen how f-structure facts are represented, this section briefly discusses how to manipulate these facts using transfer rules to produce a modified f-structure. Transfer rules are demonstrated in detail in section 3 in the context of sentence condensation. Here we just introduce the basic forms of the rules.[3] Transfer rules consist of a list of input facts, a rewrite system, and a list of output facts.[4] A very simple transfer rule that rewrites NUM sg to NUM pl is shown in (9).

(9) `NUM(%Fstr,sg) ==> NUM(%Fstr,pl).`

There is only one input fact `NUM(%Fstr,sg)`. The `%Fstr` is a variable which can match an f-structure. All mentions of the same variable in a rule must refer to the same f-structure (variables are preceded by a %). This is an obligatory rule as indicated by the rewrite symbol `==>`; an optional rule would have used the rewrite symbol `?=>`, as is demonstrated in most of the sentence condensation rules in section 3. There is only one output fact `NUM(%Fstr,pl)`. If an f-structure is found with NUM `sg`, then the rule deletes the NUM attribute and its value, and then replaces them by the attribute NUM and value `pl` listed in the output facts. The use of the same variable `%Fstr` in the input and output guarantees that the same f-structure will have the feature deleted and then added.

It is possible to have input facts that are not deleted. This is indicated by preceding the facts with a +. The rule in (10) deletes NUM `sg` and inserts NUM `pl` only when the f-structure is also third person.

(10) `NUM(%Fstr,sg), +PERS(%Fstr,3) ==> NUM(%Fstr,pl).`

The + in front of the PERS means that PERS will still be present after the rule has applied.

It is also possible to state facts which may not be present for the rule to apply. This is indicated by preceding the facts with a −. (11) is a rule which deletes NUM `sg` and inserts NUM `pl` only when the f-structure is not third person.

(11) `NUM(%Fstr,sg), -PERS(%Fstr,3) ==> NUM(%Fstr,pl).`

This use of the − operator can be seen in rule (23) in which adjuncts are deleted only if they are not negative adjuncts. This prevents *not* from being deleted in condensations, while allowing adverbs like *quickly* to be deleted.

The final important fact about the transfer rules is that they are ordered. This ordering can result in the application of some rules blocking the application of other rules or providing the necessary environment for their application. This is discussed in section 3.1.4. A simple example of this is seen in the two rules in (12).

(12) a.
```
NUM(%Fstr,sg)
   ==> NUM(%Fstr,pl).
```
b.
```
+NUM(%Fstr,pl)
   ==> NTYPE(%Fstr,count).
```

[3] There are a number of other operations that can be used in transfer rules that are not described in this paper, but are described in the XLE documentation.

[4] The list of input facts or output facts can be empty, represented by 0. This is seen in rule (36) in section 3.1.

Rule (12a) rewrites NUM sg as NUM pl; rule (12b) adds a NTYPE count feature to a plural f-structure. If (12a) is ordered before (12b), then all f-structures with a NUM feature will have the value pl for this feature and hence will end up with NTYPE count as well. If (12a) is ordered after (12b), then f-structures with NUM sg will not have NTYPE count inserted, although they will ultimately end up with NUM pl.

To summarize, in sentence condensation, the transfer component takes an f-structure as input, rewrites the f-structure facts according to an ordered set of rules, and produces an output f-structure. This f-structure is then the input to the generator which produces the string(s) corresponding to the f-structure; this string is the condensed sentence. The next section describes how the generator works; the remainder of the paper describes the sentence condensation rules in more detail.

2.3 The Generator

The sentence condensation rules manipulate the f-structure to produce new, smaller f-structures. Most of these f-structures are well formed in that the grammar can use them to generate well-formed English strings. Those that are not well formed will result in no output. Thus, it is possible to avoid generating ill-formed sentences in the condensation system by using the LFG grammar as a filter. A generator is the inverse of a parser. A parser takes a string as input and produces f-structures as output. A generator takes an f-structure as input and produces all of the strings that, when parsed, could have that f-structure as output. The generator can be useful as a component of translation (Frank 1999), sentence condensation (Riezler et al. 2003), computer assisted language learning (Butt et al. 2004), or natural language interfaces.

The XLE generator produces a packed representation of all of its output strings using a regular expression notation. An example of this is shown in (13).

(13) You {can't|cannot} {quickly print documents|print documents quickly}.

The generator can be set to only produce one output: either the shortest or the longest string. If two strings are of equal length, one of them will be chosen arbitrarily. For sentence condensation, having a single string output is usually desirable, and so the shortest string option is used.

Often it is not desirable for the generator to be the exact inverse of the parser. For instance, although the parser eliminates extra spaces between tokens, the generator should not to insert arbitrary spaces between tokens. To handle this, the generation grammar can be minimally different from the parsing grammar by changing the set of optimality marks used (Frank et al. 2001), which (dis)prefers that application of certain rules, and by changing the tokenizers or morphology that is used (Kaplan et al. 2004). These mechanisms allow minor variations in the generation grammar as needed, while still sharing as much as possible with the parsing grammar.

The remainder of this section first describes how the generator handles minor mismatches between the input f-structures produced by the sentence condensation transfer rules and the f-structures produced by the grammar when parsing. The second part of this section describes how the generator works in more detail since it is a key component in the use of LFG for sentence condensation.

2.3.1 Underspecified Input

In the default situation, the generator produces strings that, when parsed, have an f-structure that exactly matches the input f-structure. However, sometimes the generator needs to produce strings from an f-structure that matches the input except for a few features. This is true in the case of sentence condensation where the rules often affect the major parts of the f-structure, sometimes ignoring the

minor ones. For example, the passivization/activization rules discussed in section 3.1 make the subject the oblique agent and the object the subject. However, they do not manipulate the case marking of these arguments, even though case marking changes with grammatical function. If the sentence in (14a) is condensed to that in (14b) the case of *glass* must change from accusative to nominative.

(14) a. The unruly children broke the glass.

b. The glass was broken.

The XLE generator used in the sentence condensation system can be specified as to which features are deleted from or can be added to an f-structure in order to allow a string to be generated. In the case marking example, CASE would be specified as both deleted and addable since it must be changed from one value to another. The generator will freely add any features that are declared to be addable if they are consistent with the input.[5] The generator always deletes any features from the input that are specified as to be deleted, while the addition of addable features is optional.

It is also possible to use the generator in a robust mode whereby every input f-structure produces some output string. The basic idea behind robust generation in the XLE generator involves two approaches. The first is to allow the generator to relax the relationship between the input f-structure and what is generated through some special optimality theory marks (Frank et al. 2001). The second approach is to allow for a fragment grammar for generation whereby strings are generated from parts of the f-structure and then stitched together. These two robustness techniques are not used in sentence condensation because the generator is needed to filter the output of the transfer rules. Sometimes the sentence condensation rules may delete too many parts of the f-structure, in which case the output string would be uninformative. For example, the sentence condensation rules can delete conjuncts in a coordination. A sentence like (15) could be condensed to any of the possibilities in (16).

(15) Mary, Jane, and Susan arrived.

(16)
a.	Mary arrived.	d.	Mary and Jane arrived.
b.	Jane arrived.	e.	Mary and Susan arrived.
c.	Susan arrived.	f.	Jane and Susan arrived.

However, these rules might overapply, resulting in all of the conjunts (the f-structures corresponding to *Mary*, *Jane*, and *Susan*) being deleted, leaving only the verb. From such a structure, robust generation might produce an imperative (*Arrive.*) or put in a pronominal subject (*It arrives.*). However, for sentence condensation, this is not a desirable result because too much information has been lost. Instead, in such cases it is better to either produce the uncondensed string or a string from some other condensation.

[5] To control the addition of features, an optimality mark can be associated with addable attributes. Whenever an addable attribute is added to the f-structure of a generation string, then its optimality mark will also be added. The effect of this depends on the optimality mark's rank. For example, consider the additions in (i).

(i) set-gen-adds add @ALL AddedFact
set-gen-adds add @INTERNALATTRIBUTES NEUTRAL
set-gen-adds add @GOVERNABLERELATIONS NOGOOD

In this example, all of the attributes (@ALL) are first assigned the user-specified AddedFact mark. Then the internal attributes (@INTERNALATTRIBUTES), which are defined by the grammar and usually contain such features as CASE and syntactic GENDER, are assigned the NEUTRAL mark, which makes them freely addable in generation. Then the governable relations such as SUBJ and OBJ are assigned the NOGOOD mark, which means that they cannot be added. The net effect is that all of the attributes other than the internal attributes and the governable relations are assigned the AddedFact mark. These attributes can be added to the f-structure of a generation string at a cost, thus limiting the addition of features to when it is necessary to produce a string.

2.3.2 How the Generator Works

This section provides a simplified description of how the XLE generator works. A key idea in understanding generation is that it produces different equivalence classes than parsing. For every f-structure, there may be several alternative c-structures that give rise to that f-structure. For instance, parsing the sentence *Two didn't come.* and then generating from the resulting f-structure might produce the four strings in (17a) which can be packed as in (17b). This would occur if both *two* and *2* and also *n't* and *not* were canonicalized in the f-structure to the same forms. For example, the f-structure in (18) might correspond to all the strings in (17a).

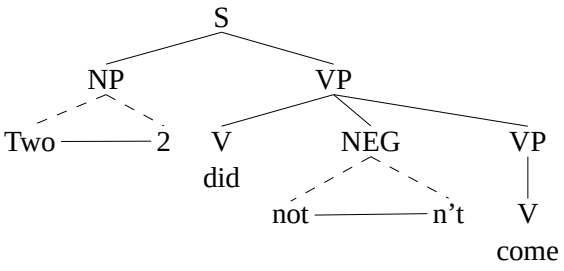
(17) a. Two didn't come. Two did not come.
2 didn't come. 2 did not come.

b. {Two|2} {didn't|did not} come.

(18)
$$\begin{bmatrix} \text{PRED} & \text{'do<XCOMP>SUBJ'} \\ \text{SUBJ} & \left[\text{PRED} \quad \text{'two'}\right]1 \\ \text{XCOMP} & \begin{bmatrix} \text{PRED} & \text{'come<SUBJ>'} \\ \text{SUBJ} & [\ 1\ ] \end{bmatrix} \\ \text{ADJUNCT} & \left\{\left[\text{PRED} \quad \text{'not'}\right]\right\} \\ \text{TENSE} & \text{past} \end{bmatrix}$$

The packed string in (17b) corresponds to a c-structure forest roughly like that in (19).

(19)



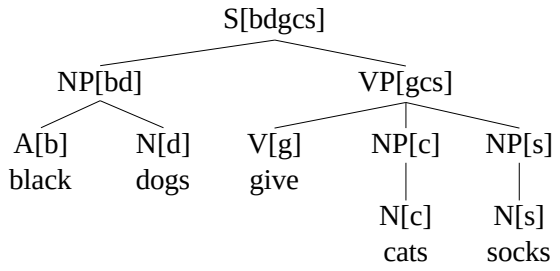
Consider the f-structure for *Two didn't come.* in (18) with links from each sub-structure to the corresponding nodes in the tree in (19). For instance, the SUBJ would map to the NP, the XCOMP to the VP over *come*, and so on. This provides a basic idea of how the generator produces alternative generations for a given f-structure. For each f-structure, the generator tries to produce all of the c-structures that are consistent with that f-structure. The result is a generation chart that is similar in spirit to a parsing chart, but with different equivalence classes.

The main performance issue with generation is to make sure that everything in the f-structure is generated at least once, and that the semantic forms are generated at most once (e.g., *the red, red dog* is not a legal regeneration of *the red dog*). Kay (1996) proposed that each edge in the generation chart would list the semantic facts that had been generated beneath it. The root edge must list all of the semantic facts in the underlying structure. If a semantic fact is missing from a root edge, then the edge is discarded.

This produces the correct result, but can be exponential in the number of adjuncts. This is because the grammar allows adjuncts to be optionally omitted, and the generator tries all paths through the grammar. If all possible combinations of optional adjuncts are considered, the result is an exponential

number of root edges. For instance, the sentence *Black dogs give white cats red socks.* might have a root edge that looks like that in (20).

(20)



At the top level, the non-adjunct predicates, *dog* [d], *give* [g], *cat* [c], and *sock* [s], are always represented. However, there could be root edges with S[dgcs], S[bdgcs], S[dgwcs], S[bdgwcs], S[dgcrs], S[bdgcrs], S[dgwcrs], or S[bdgwcrs], depending on which of the adjuncts *black*, *white*, or *red* were present. Only the last of these corresponds correctly to the input f-structure. In addition, in the grammar's phrase structure rules, the OBJ and OBJ2 are optional if extraction is allowed (e.g., for relatives and interrogatives *What did the black dogs give white cats?*), and the SUBJ can be dropped in an imperative construction (*Give white cats red socks.*). Thus, the problem is significant from a computational perspective.

Kay's 1996 solution was that in a categorial grammar, it can be locally determined when an "f-structure" becomes inaccessible. Therefore, edges that are incomplete (for instance, NP[c] and NP[s] in (20)) can be locally discarded. Unfortunately, this does not work for LFG, because f-structures can always be accessed by functional uncertainties and/or zipper unifications.[6] The XLE solution to this problem is to generate in two passes. The first pass determines which f-structures are accessible from outside each edge while ignoring the semantic facts covered. The second pass uses Kay's algorithm, but determines inaccessibility based on the first pass. The result is an efficient generator for LFG grammars.

This section has outlined the main components of the sentence condensation system: the XLE parser, transfer system, and generator. The remainder of the paper focuses on how the LFG f-structure analyses of sentences can be manipulated by the transfer rewrite rules to produce well-formed sentence condensations.

3 Manipulating F-structures

The basic idea behind the sentence condensation system is to delete or rearrange elements of an f-structure and then generate a condensed string from that new f-structure. LFG's f-structures are well suited for this task because they already encode much of the work for the system.

For example, elements of a sentence that are inessential for condensation purposes often coincide with ADJUNCTs in the f-structure. This contrasts with string or tree representations of sentences in which it can be extremely difficult to determine what the adjuncts are. However, in LFG, adjuncts receive a special f-structure function ADJUNCT and hence a simple rule can be written that deletes them, as in (21). (As discussed in section 2.2.2, `?=>` indicates an optional rule.)

(21) `+ADJUNCT(%Main,%AdjSet), in_set(%Adjunct,%AdjSet) ?=> 0.`

[6] Zipper unification occurs during the unification of f-structures when there are sequences of nested sub-structures that must be unified because they have common features. See Maxwell and Kaplan 1996 for processing details and Bresnan et al. 1982 for how this applies to cross-serial dependencies in Dutch.

This rule would apply to the f-structure in (22b) for sentence (22a), resulting in the possible outputs in (22c) since the rules apply optionally in the system. (The f-structures which the rule variables can match are shown in italics; these variable names are not part of the f-structure input; cf. (6).)

(22) a. Mary arrived yesterday in her car.

b.
$$\begin{bmatrix} \text{PRED} & \text{'arrive<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \end{bmatrix} \\ \text{ADJUNCT} & \left\{ \begin{bmatrix} \text{PRED} & \text{'yesterday'} \end{bmatrix}_{\%Adjunct} \begin{bmatrix} \text{PRED} & \text{'in<OBJ>'} \\ \text{OBJ} & \text{[her car]} \end{bmatrix}_{\%Adjunct} \right\}_{\%AdjSet} \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

c. No deletions: Mary arrived yesterday in her car.
Both adjuncts deleted: Mary arrived.
First adjunct deleted: Mary arrived in her car.
Second adjunct deleted: Mary arrived yesterday.

The detailed feature space of LFG f-structures is particularly useful in this example because there are some adjuncts that should never be deleted, namely negatives. Because such adjuncts are assigned a feature ADJUNCT-TYPE `negative` in the grammar, a restriction can be placed on rule (21) such that it does not apply to adjuncts with this feature. This is shown in (23); the modified rule in (23a) will allow the deletions in (22c), but not that in (23b).

(23) a.
```
ADJUNCT(%Main,%AdjSet), in_set(%Adjunct,%AdjSet),
−ADJUNCT-TYPE(%Adjunct,negative)
?=> 0.
```

b. Mary did not arrive. $\Longrightarrow$ *Mary arrived.

The deletion of adjuncts is a common strategy in sentence condensation systems, albeit one that is difficult to implement without something like LFG's f-structures. However, f-structures allow for more complex manipulations which can provide for more natural sounding condensations. For example, the LFG f-structures provided by the grammar allow for rules which turn clauses with *you should* ... into imperatives, remove phrases like *it is clear that* ... leaving only the subordinate clause, decleft sentences, and remove conjuncts from coordinations. More than one rule can apply to a given sentence, and rule ordering allows for feeding and bleeding. Here we discuss two of these more complex rules: passivization/activization (section 3.1) and factive verb deletion (section 3.2).

3.1 Passivization and Activization

There are a number of ways to manipulate active and passive sentences. One is to take passive sentences with *by* phrases and create their active counterparts, as in (24a). Another is to take a passive without a *by* phrase and turn it into an active with a *they* subject, as in (24b). A third is to passivize an active sentence, as in (24c). Although these do not always result in fewer words in the sentence, they may simplify the content for the reader.

(24) a. The town was flooded by torrential rains. $\Longrightarrow$ Torrential rains flooded the town.

b. The car was pushed off the tracks. $\Longrightarrow$ They pushed the car off the tracks.

c. Torrential rains flooded the town. $\Longrightarrow$ The town was flooded.

Rules for all three alternations appear in our system and we discuss them here in detail.

3.1.1 Activization of passives with *by* phrases

The rule for the alternation in (24a) is shown in (25) and would apply to an f-structure like that in (26). This rule takes an oblique agent phrase in a passive and makes it the subject while the subject is demoted to object. This is basically the reverse of the well-know LFG passivization lexical rule.

(25)
```
PASSIVE(%Main,+), SUBJ(%Main,%Subj), OBL-AG(%Main,%OblAg)
PFORM(%OblAg,by_), PTYPE(%OblAg,nosem) ?=>
SUBJ(%Main,%OblAg), OBJ(%Main,%Subj), PASSIVE(%Main,−).
```

(26)
$$\begin{bmatrix}
\text{PRED} & \text{'flood<OBL-AG,SUBJ>'} \\
\text{SUBJ} & \left[\text{PRED} \quad \text{'town'}\right]_{\%Subj} \\
\text{OBL-AG} & \begin{bmatrix}
\text{PRED} & \text{'rain'} \\
\text{PFORM} & \text{by_} \\
\text{PTYPE} & \text{nosem} \\
\text{ADJUNCT} & \left\{\left[\text{PRED} \quad \text{'torrential'}\right]\right\}
\end{bmatrix}_{\%OblAg} \\
\text{PASSIVE} & + \\
\text{TENSE} & \text{past}
\end{bmatrix}_{\%Main}$$

Consider the rule in (25) in detail. Each of the facts before the rewrite symbol (?=>) exists in the f-structure in (26): there is a PASSIVE + feature, a SUBJ, and a OBL-AG with the correct PFORM and PTYPE. None of these facts is preceded by a +; this means that each of them will be deleted in the f-structure. In their place, the facts after the rewrite symbol will be inserted into the f-structure. So, there will be a new SUBJ, a new value for PASSIVE, and an OBJ will be created. The resulting f-structure will be as in (27). When this new f-structure is run through the generator, it will produce the active sentence *Torrential rains flooded the town.*

(27)
$$\begin{bmatrix}
\text{PRED} & \text{'flood<SUBJ,OBJ>'} \\
\text{SUBJ} & \begin{bmatrix}
\text{PRED} & \text{'rain'} \\
\text{ADJUNCT} & \left\{\left[\text{PRED} \quad \text{'torrential'}\right]\right\}
\end{bmatrix}_{\%OblAg} \\
\text{OBJ} & \left[\text{PRED} \quad \text{'town'}\right]_{\%Subj} \\
\text{PASSIVE} & - \\
\text{TENSE} & \text{past}
\end{bmatrix}_{\%Main}$$

3.1.2 Activization of short passives with generic *they* subject

Next consider the rule in (29) which performs the condensation in (24b), repeated as (28). In this example, a passive sentence becomes an impersonal active sentence with the generic subject *they*.

(28) The car was pushed off the tracks. $\Longrightarrow$ They pushed the car off the tracks.

(29)
```
PASSIVE(%Main,+), SUBJ(%Main,%Subj), arg(%Main,1,NULL) ?=>
PASSIVE(%Main,−), OBJ(%Main,%Subj),
SUBJ(%Main,%NewSubj), arg(%Main,1,%NewSubj),
PRED(%NewSubj,pro), NUM(%NewSubj,pl), PERS(%NewSubj,3),
PRON-FORM(%NewSubj,they).
```

First consider the input facts that appear before the rewrite symbol. They require a passive verb with a subject. In addition, they require that the first argument of the predicate be NULL, as opposed to being an OBL-AG. The arguments of a predicate are referenced by the built-in predicate `arg` which takes three arguments: an f-structure, a number indicating which argument it is, and a value for this argument. In this example, the f-structure is the one containing the verb (`%Main`), the argument number is 1 since it is the first argument of the verb that we are interested in, and the value of this is NULL.[7] Looking at the facts after the rewrite symbol, PASSIVE is changed to $-$ and an OBJ is created from the original subject. Then a new SUBJ is created. All of the features are provided for this new subject: predicate, number, person, form. To guarantee that this subject appears in the correct argument slot, another mention of `arg` is used to create this new argument as the first argument of the main predicate.[8]

The f-structure in (30) will be rewritten as that in (31) by rule (29).

(30)
$$\begin{bmatrix} \text{PRED} & \text{'push<NULL,SUBJ>'} \\ \text{SUBJ} & \left[\text{PRED} \quad \text{'car'}\right]_{\%Subj} \\ \text{ADJUNCT} & \left\{\left[\begin{matrix}\text{PRED} & \text{'off<OBJ>'} \\ \text{OBJ} & \left[\text{PRED} \quad \text{'track'}\right]\end{matrix}\right]\right\} \\ \text{PASSIVE} & + \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

[7] This value is often another f-structure. To refer to the second argument of the passive verb in (29), `arg(%Main,2,%Subj)` would be used.

There is another built-in predicate `nonarg` which refers to non-thematic arguments. For a predicate like *'want*<SUBJ,XCOMP>OBJ' for sentences like *John wants Mary to leave.*, the arguments can be referred to as in (i).

(i)
```
SUBJ    arg(%Main,1,%Subj)
XCOMP   arg(%Main,2,%XComp)
OBJ     nonarg(%Main,1,%Obj)
```

[8] The rule in (25) did not have to refer to `arg` because both the subject and the oblique agent were mentioned in the original predicate in the correct order. All that was done was to change the names. Putting in a call to `arg` would not cause problems, but it is not necessary. It is only when argument slots are being added, such as when NULL is rewritten as an overt argument, that `arg` is needed.

(31)
$$\begin{bmatrix} \text{PRED} & \text{'push<SUBJ,OBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'pro'} \\ \text{NUM} & \text{pl} \\ \text{PERS} & 3 \\ \text{PRON-FORM} & \text{they} \end{bmatrix}_{\%NewSubj} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'car'} \end{bmatrix}_{\%Subj} \\ \text{ADJUNCT} & \left\{ \begin{bmatrix} \text{PRED} & \text{'off<OBJ>'} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'track'} \end{bmatrix} \end{bmatrix} \right\} \\ \text{PASSIVE} & + \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

3.1.3 Passivization

Finally consider the passivization example repeated in (32). The active sentence can be rewritten as a passive. This passive can either have the active subject as an oblique agent or have it deleted. The first option, in which the subject becomes an oblique agent, will not result in a condensed sentence. However, we show the rule for this in (33) since it can be used to feed and bleed other rules (see section 3.1.4). We then discuss a second rule which deletes oblique agents regardless of their source.

(32) Torrential rains flooded the town.
$\Longrightarrow$ The town was flooded by torrential rains.
$\Longrightarrow$ The town was flooded.

(33)
```
PASSIVE(%Main,-), SUBJ(%Main,%Subj), OBJ(%Main,%Obj) ?=>
PASSIVE(%Main,+), SUBJ(%Main,%Obj),
OBL-AG(%Main,%Subj), PFORM(%Subj,by_), PTYPE(%Subj,nosem).
```

The rule in (33) is basically the inverse of the one in (25). The input facts are a non-passive f-structure with a subject and an object. These are deleted and the PASSIVE feature it changed to +, a new subject is created from the object's f-structure, and an oblique agent is created from the subject's f-structure. In addition, the new oblique agent is marked as having the features needed for the *by* found in *by* phrases for oblique agents. The passivization rule in (33) can apply to the f-structure in (34) to produce an f-structure as in (35).

(34)
$$\begin{bmatrix} \text{PRED} & \text{'flood<SUBJ,OBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'rain'} \\ \text{ADJUNCT} & \left\{ \begin{bmatrix} \text{PRED} & \text{'torrential'} \end{bmatrix} \right\} \end{bmatrix}_{\%Subj} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'town'} \end{bmatrix}_{\%Obj} \\ \text{PASSIVE} & - \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

(35)
$$\begin{bmatrix} \text{PRED} & \text{'flood<OBL-AG,SUBJ>'} \\ \text{SUBJ} & \left[\text{PRED} \quad \text{'town'}\right]_{\%Obj} \\ \text{OBL-AG} & \begin{bmatrix} \text{PRED} & \text{'rain'} \\ \text{PFORM} & \text{by\_} \\ \text{PTYPE} & \text{nosem} \\ \text{ADJUNCT} & \left\{\left[\text{PRED} \quad \text{'torrential'}\right]\right\} \end{bmatrix}_{\%Subj} \\ \text{PASSIVE} & + \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

To obtain a version of the sentence which does not have an oblique agent, a separate rule can be applied that deletes the oblique agent. This rule does not discriminate between situations in which the oblique agent is the result of the passivization rule in (33) and ones in which the original sentence was a passive.

(36) `OBL-AG(%Main,%OblAg), arg(%Main,%%,%OblAg) ?=> 0.`

The rule in (36) can apply to the f-structure in (35) which was the output of the passivization rule (the variable `%OblAg` in (36) will match the f-structure labelled *%Subj* in (35)); the result is the structure in (37). The input facts are the OBL-AG and its argument position in the verbal predicate. Both of these are deleted. If the `arg(%Main,%%,%OblAg)` fact was not deleted, then the generator would not be able to generate from the resulting f-structure because that f-structure would have an argument slot with nothing to fill it.[9] The `0` after the rewrite symbol indicates that the input facts are deleted; this is exactly as in the adjunct deletion rule in (23).

(37)
$$\begin{bmatrix} \text{PRED} & \text{'flood<NULL,SUBJ>'} \\ \text{SUBJ} & \left[\text{PRED} \quad \text{'town'}\right] \\ \text{PASSIVE} & + \\ \text{TENSE} & \text{past} \end{bmatrix}_{\%Main}$$

3.1.4 Feeding and Bleeding

Consider the four rules we discussed above for activization and passivization, shown in (38). The transfer system used in sentence condensation orders the rules that apply. The four rules here need to be ordered in such a way that they correctly feed and bleed each other.

(38) a. Passive + oblique agent $\Longrightarrow$ active: rule (25)

b. Passive without oblique agent $\Longrightarrow$ active with *they* subject: rule (29)

c. Active $\Longrightarrow$ passive with oblique agent: rule (33)

d. Delete oblique agent: rule (36)

[9] The %% is a variable that matches any value. So, in (36) the rule states that it is unimportant what number the argument was in the predicate. This use of anonymous variables is also seen in rule 42 in which the feature COMP-FORM is deleted regardless of its value.

Feeding occurs when the application of one rule produces the structure needed for the application of another rule. For example, consider a sentence like (39).

(39) The enemy invaded the town.

Rule (38c) can apply to (39) to create a passive version. This creates (feeds) the environment for rule (38d) which deletes the oblique *by phrase*. This process is shown in (40).

(40) a. The enemy invaded the town.

b. The town was invaded by the enemy. (Rule 38c)

c. The town was invaded. (Rule 38d)

Bleeding occurs when the application of one rule destroys the environment for another. Consider the sentence in (40b). If rule (38a) applies creating an active sentence, then rule (38d) which deletes the oblique agent cannot apply.

In addition, each rule was written as an optional rule (?=>) and not an obligatory one (==>). Whether or not a rule applies will affect which further rules can apply to the f-structure. For example, suppose that the rules in (38) were in the opposite order. If rule (38d) applies to (40b) deleting the *by* agent, then rule (38b) can apply to create a *they* active. If the option is chosen where the rule does not apply and hence the *by* agent remains, then rule (38a) can apply to create a standard active sentence. As such, it is important to consider the ordering of the rules and which ones should be obligatory and which optional. Almost all the rules in the sentence condensation system are optional. In the case of the activization/passivization rules, this creates significant feeding and bleeding, even with the rule ordering. However, in applications like machine translation, most of the rules are obligatory, although feeding and bleeding issues still arise and can be exploited by the system.

3.2 Factive Verb Deletion

Next consider the condensation rules for factive verb deletion. Sentences with factive verbs can be condensed by keeping just the complement of the verb, as in (41a).

(41) a. They realized that Mary left yesterday. $\Longrightarrow$ Mary left yesterday.

b. They did not realize that Mary left yesterday. $\Longrightarrow$ Mary left yesterday.

The LFG f-structure analysis, shown in (43) for (41a), in conjunction with lexicalization of the condensation rules allows for the simple formulation of such rules, as in (42).

(42)
```
PRED(%Main,realize), COMP(%Main,%Comp), COMP-FORM(%Comp,%%)
?=> 0.
```

(43)
$$\begin{bmatrix} \text{PRED} & \text{'realize<SUBJ,COMP>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'pro'} \end{bmatrix} \\ \text{COMP} & \begin{bmatrix} \text{PRED} & \text{'leave<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \end{bmatrix} \\ \text{ADJUNCT} & \left\{ \begin{bmatrix} \text{PRED} & \text{'yesterday'} \end{bmatrix} \right\}_{\%Comp} \\ \text{TENSE} & \text{past} \\ \text{COMP-FORM} & \text{that} \end{bmatrix}_{\%Main} \\ \text{TENSE} & \text{past} \end{bmatrix}$$

The input facts in (42) match the f-structure in (43). The 0 after the rewrite symbol states that nothing is added. The desired output f-structure is that in (44) which corresponds to the sentence *Mary left*.

(44)
$$\left[\begin{array}{ll} \text{PRED} & \text{'leave<SUBJ>'} \\ \text{SUBJ} & \left[\text{PRED} \quad \text{'Mary'}\right] \\ \text{ADJUNCT} & \left\{\left[\text{PRED} \quad \text{'yesterday'}\right]\right\} \\ \text{TENSE} & \text{past} \end{array}\right]_{\%Comp}$$

3.2.1 Deleting F-structures

Looking more carefully at the rule in (42) there are a number of facts in the input f-structure in (43) that are not explicitly mentioned in the rule and hence will not be explicitly deleted. These facts from the main f-structure *%Main* are shown in (45).

(45)
SUBJ	$\Longleftrightarrow$	`SUBJ(%Main,%Subj)`
[PRED 'pro']	$\Longleftrightarrow$	`PRED(%Subj,pro)`
TENSE past	$\Longleftrightarrow$	`TENSE(%Main,past)`

These facts will be deleted by the transfer system because they are no longer connected to the top level f-structure. As such, it is unnecessary to mention every feature that might appear in an f-structure that is to be deleted or in the subsidiary f-structures of such an f-structure. This is essential because it is generally impossible to predict what additional information will occur in such structures. For example, the deleted verb *realize* might have had any number of adjunct modifiers, as in (46), all of which are to be deleted.

(46) Upon their arrival, they quickly realized that Mary left yesterday.

The rule in (42), however, also leaves the f-structure that was the value COMP (*%Comp*) disconnected from the top level f-structure. This is not what was intended. Instead, the intention was to have the f-structure of the COMP become the top level f-structure. This can be done by initially setting the original top level f-structure to a special fact called `root` and then redefining `root` to have the value of the COMP's f-structure.[10]

3.2.2 Templates for Lexicalized Rules

The factive verb deletion rule is lexicalized in that it only occurs with certain lexical items: not all verbs with *that* complements presuppose their complement. It would be possible to repeat the rule in (42) as many times are there are factive verbs, replacing the value of the verbal predicate in `PRED(%Main,realize)`. However, this misses a generalization and can result in maintenance difficulties. Instead, the rule in (42) can be defined as a template and each factive verb will call this template. The template version of (42) is shown in (47) with sample calls in (48). The @ indicates a call to a template.[11]

[10]The initial setting of `root` is done by placing the following call at the beginning of the transfer rules.

```
:- set_transfer_option(extra,[cf(1, root(root, var(0)))]).
```

[11]Templates must be defined before they are called, otherwise the transfer system will not correctly rewrite them as rule applications.

(47)
```
factive_verb_deletion(%Verb) ::
PRED(%Main,%Verb), COMP(%Main,%Comp), COMP-FORM(%Comp,%%)
?=> 0.
```

(48)
```
@factive_verb_deletion(realize).
@factive_verb_deletion(know).
```

The template in (47) takes one argument, the value of the PRED of the factive verb. This value is then substituted into the input facts of the template, resulting in a rule application similar to that in rule (42). For example, the two template calls in (48) will result in two versions of the rule, one for *realize* and one for *know*, as if the two rules in (49) had been included in the rule set. Large lists of verbs can be added by using templates in this way.

(49) a.
```
PRED(%Main,realize), COMP(%Main,%Comp), COMP-FORM(%Comp,%%)
?=> 0.
```

b.
```
PRED(%Main,know), COMP(%Main,%Comp), COMP-FORM(%Comp,%%)
?=> 0.
```

Using templates for rules that are triggered by particular lexical items can be very valuable. In the sentence condensation system, there are a number of these rules, including ones for deleting group terms (50a), keeping only the complements of certain adjectives (50b), and intransitivization (50c).

(50) a. A set of tools was found. $\Longrightarrow$ Tools were found.
```
@delete_group(set).
@delete_group(bunch).
@delete_group(group).
```

b. It is clear that they left. $\Longrightarrow$ They left.
```
@delete_copular_adjective(clear).
@delete_copular_adjective(obvious).
@delete_copular_adjective(evident).
```

c. They broke the glass. $\Longrightarrow$ The glass broke.
```
@intransitivization(break).
@intransitivization(decrease).
@intransitivization(increase).
```

4 Conclusions

In Riezler et al. 2003, we presented an application of ambiguity packing and stochastic disambiguation techniques for LFG to the domain of sentence condensation. The system incorporated a linguistic parser/generator for LFG, a transfer component for f-structure reduction using the rules described in this paper, and a maximum-entropy model for stochastic output selection. Overall summarization quality of the proposed system was tested on a small corpus and proved to be state-of-the-art, with guaranteed grammaticality of the system output due to the use of the LFG parser/generator.

In this paper, we described the types of sentence condensation rules used in the system in detail. We showed how the distinctions made in LFG f-structures as to grammatical functions and features make it possible to state simple but accurate rules to create smaller, well-formed f-structures from

which the condensed sentence can be generated. In addition, we described the elements of the XLE system that make it possible to parse, transfer, and then generate the structures needed for accurate sentence condensation.

References

Bresnan, Joan, Ron Kaplan, Stanley Peters, and Annie Zaenen. 1982. Cross-serial dependencies in Dutch. *Linguistic Inquiry*, 13(4):613–35.

Butt, Miriam, Helge Dyvik, Tracy Holloway King, Hiroshi Masuichi, and Christian Rohrer. 2002. The Parallel Grammar Project. In *Proceedings of COLING-2002 Workshop on Grammar Engineering and Evaluation*. pp. 1-7.

Butt, Miriam, Rafiq Khader, and Tracy Holloway King. 2004. Deep CALL Grammars: The LFG-OT Experiment. Jahrestagung der Deutschen Gesellschaft dür Sprachwissenschaft, Mainz, Germany.

Butt, Miriam, Tracy Holloway King, Maria-Eugenia Nino, and Frederique Segond. 1999. *A Grammar Writer's Cookbook*. Stanford: CSLI Publications.

Frank, Anette. 1999. From Parallel Grammar Development towards Machine Translation. In *Proceedings of MT Summit VII*. pp. 134-142.

Frank, Anette, Tracy Holloway King, Jonas Kuhn, and John Maxwell. 2001. Optimality Theory style constraint ranking in large-scale LFG grammars. In Peter Sells, editor, *Formal and Empirical Issues in Optimality Theory*. Stanford: CSLI Publications. pp. 367-397.

Kaplan, Ron, John T. Maxwell III, Tracy Holloway King, and Richard Crouch. 2004. Integrating Finite-state Technology with Deep LFG Grammars. *Proceedings of the Workshop on Combining Shallow and Deep Processing for NLP (ESSLLI)*.

Kay, Martin. 1996. Chart Generation. In *Proceedings of the 34th conference on Association for Computational Linguistics*, pp. 200-204.

Knight, Kevin, and Daniel Marcu. 2000. Statistics-based summarization - step one: Sentence compression. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*.

Maxwell, John, and Ron Kaplan. 1991. A Method for Disjunctive Constraint Satisfaction. *Current Issues in Parsing Technologies*.

Maxwell, John, and Ron Kaplan. 1993. The Interface Between Phrasal and Functional Constraints. *Computational Linguistics* 19:4.

Maxwell, John, and Ron Kaplan. 1996. An Efficient Parser for LFG. In *Proceedings of the First LFG Conference*. CSLI On-line Publications.

Riezler, Stefan, Tracy Holloway King, Richard Crouch, and Annie Zaenen. 2003. Statistical sentence condensation using ambiguity packing and stochastic disambiguation methods for Lexical-Functional Grammar. In *Proceedings of the Human Language Technology Conference and the 3rd Meeting of the North A merican Chapter of the Association for Computational Linguistics (HLT-NAACL'03)*.

Riezler, Stefan, Tracy Holloway King, Ron Kaplan, Richard Crouch, John Maxwell, and Mark Johnson. 2002. Parsing the Wall Street Journal using a Lexical-Functional Grammar and Discriminative Estimation Techniques. *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Witbrock, M. and V. Mittal. 1999. Ultra-summarization: A statistical approach to generating highly condensed non-extractive summaries. In *Proceedings of the 22nd ACM SIGIR Conference on Research and Development in Information Retrieval*.