Compilation of Unification Grammars with Compositional Semantics to Speech Recognition Packages

Johan Bos
Institute for Communicating and Collaborative Systems
Division of Informatics, University of Edinburgh
2 Buccleuch Place, Edinburgh EH8 9LW, Scotland, UK
Johan.Bos@ed.ac.uk

Abstract

In this paper a method to compile unification grammars into speech recognition packages is presented, and in particular, rules are specified to transfer the compositional semantics stated in unification grammars into speech recognition grammars. The resulting compiler creates a context-free backbone of the unification grammar, eliminates left-recursive productions and removes redundant grammar rules. The method was tested on a medium-sized unification grammar for English using Nuance speech recognition software on a corpus of 131 utterances of 12 different speakers. Results showed no significant computational overhead with respect to speech recognition performances for speech recognition grammar with compositional semantics compared to grammars without.

1 Introduction

This paper presents a method to generate speech recognition packages that produce generic domain independent logical forms. Therefore, no subsequent post-processing is needed (apart from β-conversion) to derive the logical form from a string by use of a parser. Moreover, the proposed method is domain independent, because it shows how to construct generic logical forms in speech recognition packages in a compositional way using the lambda calculus.

In particular, a system is presented that compiles medium-sized linguistic unification grammars (UG) into the *Grammar Specification Language* (GSL) that Nuance[1] speech recognition software requires before compiling it to finite state machines making up the language model. The method described in this paper has close points of contact with recent research on compiling domain independent linguistically motivated grammars into GSL (Rayner et al., 2000; Rayner et al., 2001b; Rayner et al., 2001a). Language models based on GSL are relatively cheap to generate and therefore are an interesting alternative to statistical models, which are relatively expensive to produce and require a relatively large corpus. Compiling from UG to GSL finds also motivation in the fact that UGs are much easier to maintain.

The compiling method described in this paper extends previous work (mentioned above) in this tradition by adding a genuine and completely general semantic component to GSL grammars. There are several challenges to meet: the language models for Nuance do not allow left-recursion in the grammar rules and there is no support for feature unification. The basic idea is to map this unification grammar into GSL, the format that Nuance requires before compiling it to finite state machines, making up the language model for the speech recogniser. This compilation includes eliminating left recursion and provides means for a generic compositional semantics. This paper discusses the requirements for such a compilation and proposes and implements a solution.

First, an overview of the architecture of the compiler, named UNIANCE, is given, and unification grammars and GSL (Section 2) are introduced. Then each system component is described in detail in Sections 3–7. The results obtained with the compiler are evaluated with respect to practical speech recognition and presented in Section 8.

2 System Overview

Input to the UNIANCE compiler are phrase structure rules of the form LHS $\rightarrow$ RHS, where

[1] See: www.nuance.com.

```
[cat:np,case:_,num:N,per:3,refl:no,sem:apply(X,Y)] --> [cat:det,count:C,num:N,sem:X],
                                                      [cat:noun,count:C,num:N,sem:Y].

[cat:noun,count:yes,num:sg,sem:lambda(X,radio(X))] --> [lex:radio].

[cat:noun,count:yes,num:sg,sem:lambda(X,car(X))] --> [lex:car].
```

Figure 1: Examples of unification grammar rules.

LHS (the left-hand side) is a single (non-terminal) category, and RHS (the right-hand side) a sequence of (terminal or non-terminal) categories. Non-terminal categories consist of a category symbol C annotated with a finite (possibly empty) set of pairs of distinct features and values, as shown in Figure 1. Terminal categories define lexical items.

A restricted form of unification grammars is considered, where features values can only be instantiated with atomic values (except for one special *slash* feature).[2] As usual in unification grammars, values for features can be left unspecified, and values of different instances of features can be constrained to have the same value (by unification). Hence feature unification constrains possible derivations expressed in the grammar.

Most features constrain syntactic derivations and have a finite set of values. The feature `sem` is used to build up semantic representations for the parsed utterances. A completely general and standard compositional semantics will be used. Expressions of the form `lambda(X,F)` denote lambda abstraction, where `X` is the variable abstracted over, and `F` a formula. Expressions of the form `apply(A,B)` denote functional application, where `A` is the functor, and `B` plays the role of argument. Some examples of the use of these expressions are shown in Figure 1. Note that in each production of a UG, the semantic values of `sem` features in RHS categories are always variables, and that the value of the `sem` feature of the LHS is (normally) expressed in terms of the values of the `sem` feature of the RHS.

The job of the compiler is to output a grammar in GSL format describing the same fragment as the input grammar. This is a challenging task because GSL has a number of restrictions in its expressiveness: there is no support for features or feature unification, and left-recursive rules (productions) are not allowed.

GSL is a form of context-free grammar where terminal symbols start with lowercase symbols, non-terminals with uppercase symbols, round brackets indicate sequence, and square brackets alternatives. An additional technique incorporated in GSL, *slot-filling*, provides means to return information (other than the recognised string) and is often used in speech applications to derive domain-specific interpretations.

```
NpCaseNomNumPlPer3ReflNo
[(DetCountYesNumPl:a NounCountYesNumPl:b)
{return(strcat("apply(" strcat(strcat($a strcat("," $b)) ")")))}]

NpCaseNomNumSgPer3ReflNo
[(DetCountYesNumSg:a NounCountYesNumSg:b)
{return(strcat("apply(" strcat(strcat($a strcat("," $b)) ")")))}]

NpCaseObjNumPlPer3ReflNo
[(DetCountYesNumPl:a NounCountYesNumPl:b)
{return(strcat("apply(" strcat(strcat($a strcat("," $b)) ")")))}]

NpCaseObjNumSgPer3ReflNo
[(DetCountYesNumSg:a NounCountYesNumSg:b)
{return(strcat("apply(" strcat(strcat($a strcat("," $b)) ")")))}]

NounCountYesNumSg
[
 (radio) return("lambda(X,radio(X))")
 (car) return("lambda(X,car(X))")
]
```

Figure 2: Some example GSL rules.

Figure 2 shows the GSL equivalent for the unification grammar rules in Figure 1. In this example one slot was assigned to each category to store its semantic representation, which is passed up to mother nodes by instances of `return/1` while using the GSL built-in `strcat/2` function to generate functional application operations using string concatenation. The compiler generates such equivalent GSL grammars from unification grammars.

The compiler, implemented in Prolog, functions as a pipeline of different components in the following order: feature instantiation, elim-

[2] Another restriction that imposed on the unification grammar is that RHS need to be non-empty. There is a standard way of eliminating ϵ-productions from a context-free grammar (Aho et al., 1996), and it is planned to integrate such a procedure in future version of the compiler.

inating left recursion, reducing redundant rules, packing and compressing the remaining grammar rules, and finally annotating rules with semantic operations (Figure 3). These components will be discussed in the remainder of this paper.

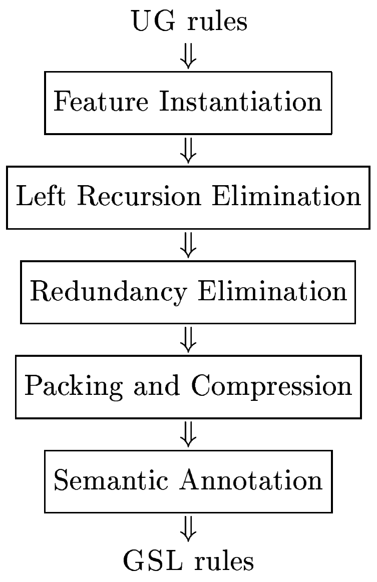

Figure 3: The architecture of UNIANCE.

3 Feature Instantiation

This component of the compiler reads in the unification grammar (including lexical rules) and converts all rules into the compiler's internal format. The primary task of this component is instantiating features with all of their possible values. In other words, it creates a context-free backbone of the input unification grammar. This process is carried out only for features with a finite number of possible values. Hence, the feature `sem` is ignored in this part of the compilation. Feature instantiation is implemented by collecting the range of feature values followed by a top-down traversal of rule instantiation.

In an initial step, all grammar and lexical rules are inspected and for each category encountered the range of possible features and their values are collected. This gives all possible features used but not necessarily all possible values. So in a consequent step, the range of possible values is expanded by examining different values for features of related categories. Categories C_1 and C_2 are related if C_1 and C_2 denote the same categories, if C_1 is a parent of C_2, or if C_2 is a parent of C_1. For instance, if the initial step yields for the feature F a value V_1 for category C_1, and a value V_2 for category C_2, and C_1 and C_2 are related, then V_2 is also a value for F with respect to C_1.

Using this information, the final stage of feature instantiation takes a grammar rule, instantiates all features with possible values (for those features with variables as values) and verifies each assignment by checking whether the instantiation of the rule is supported by the current set of grammar rules. This is done in a top-down fashion, by recursing on the categories of the right-hand side of each rule. Finally, the feature instantiation component passes the instantiated rules to the component that eliminates left recursive rules. The rules have now been translated in a format $\mathrm{C}_0 \rightarrow \mathrm{C}_1...\mathrm{C}_n$, where C_i is of the form cat(A,F,X) and A is a category symbol, F is a finite (possible empty) set of instantiated feature value pairs, and X the semantic representations. The result is a CFG backbone which can (almost) be compiled into a GSL-style grammar.

However, there seem some clouds on the horizon. GSL grammars with left-recursive rules cannot be compiled into Nuance speech recognition grammars so they need to be replaced by equivalent right-recursive ones. Furthermore, if one is able to do this, one need to do this for the `sem` feature as well, in order to obtain the right meaning representations on the transformed trees. The next section shows how to deal with this problem.

4 Eliminating Left Recursion

Left-recursive rules are common in linguistically motivated grammars to express coordination or modification. Since GSL does not allow left-recursive rules, there is a need to introduce a way to transform grammars containing left-recursive productions into grammars without. There is a standard way of eliminating left-recursion from a context free grammar (Aho et al., 1996), as long as the grammar contains no cycles (rules of the form $C \rightarrow C$, which of course do not appear in any sensible linguistic grammar), and no ϵ-productions. This is also the method used in this paper.

The current version of the compiler only considers *immediate* cases of left recursion. The standard way of dealing with this is to note that if there is a production rule expressing left recursion, there must be some production with the same left hand side that is not left recursive. What one needs to do is rewrite the left recursive production in terms of the production that does not have any left recursion, which can be done by replacing left-recursive pairs of productions $A \rightarrow AB$, $A \rightarrow C$ by $A \rightarrow CA'$, $A' \rightarrow BA'$ and $A' \rightarrow \epsilon$. A similar transformation, but not introducing ϵ productions, can be formulated as replacing left-recursive pairs of productions $A \rightarrow AB$, $A \rightarrow C$ by $A \rightarrow CA'$, $A \rightarrow C$, $A' \rightarrow B$, and $A' \rightarrow BA'$. Although the first method produces less rules than the second method, in terms of implementation the latter is preferred because it is easier to transfer the results of the transformation into GSL (no empty productions are generated with the second method).

The standard way of eliminating left-recursive features deals with the context-free backbone of the grammar, but not with the compositional semantics, i.e., the values of the `sem` feature. A category with name A and features F is left-recursive if there are rules of the form cat(A,F,X) $\rightarrow$ cat(A,F,Y) $C_1 ... C_n$. Two elimination rules are defined: one that covers the non-recursive grammar rules for a recursive category, and one that covers the left-recursive cases.

LR Elimination Rule I

For each left-recursive category, and for all non-recursive grammar productions of the form

cat(A,F,X) $\rightarrow$ $C_1 ... C_n$

extend the grammar with productions of the form:

cat(A,F,apply(Z,X))
$\rightarrow$ $C_1 ... C_n$ cat(new,[rec:A|F],Z)

LR Elimination Rule II

For each left-recursive category, replace all recursive productions of the form

cat(A,F,X)
$\rightarrow$ cat(A,F,Y) $C_1 ... C_n$

by the following two productions:

cat(new,[rec:A|F],lambda(Y,apply(Z,X)))
$\rightarrow$ $C_1 ... C_n$ cat(new,[rec:A|F],Z)

cat(new,[rec:A|F],lambda(Y,X))
$\rightarrow$ $C_1 ... C_n$

The LR elimination rules introduce new categories and transform the semantic operations in such a way that the semantic representations yielded by the transformed rules are logically equivalent to their left-recursive counterpart. This is done by what is very similar to *type-shifting* in a Montagovian semantics: introducing lambda abstraction (LR Elimination Rule II) and postponing application (LR Elimination Rule I). It ensures that the order of semantic combinations in the original grammar is maintained in the transformed grammar. Elimination Rule II is the most interesting one. By abstracting over Y (the semantic value of the recursive category in the RHS), it guarantees that any occurrences of Y in X (the semantic value of the recursive category in the LHS) are bound, no matter how the semantics for X is defined in terms of the semantic values for the RHS categories.

Cases of indirect left recursion can be dealt with by using the algorithm provided by Aho et al. (1996), which eliminates *all* left recursion from a grammar. This algorithm iterates on all productions where nonterminals are part of the left hand side, by replacing all nonterminals in the right hand side by its corresponding production and then applying the transformation for immediate left-recursive rules as presented above to the resulting new production.

5 Removing Redundant Productions

Grammar rules might turn out to be redundant because certain lexical items in the grammar do not appear in the selected application domain or the grammar itself might be ill-formed. There can be two reasons for a production to be redundant:

1. For a non-terminal member C of the RHS there is no production where C appears as LHS (redundant RHS productions);

2. A LHS category, but not the beginner category, does not appear as member of the

RHS of any other production (redundant LHS productions).

Productions that violate any of the constraints above are removed from the grammar. Of course, removing rules might cause other ones to be irrelevant. So this elimination process will continue until a fixed point is reached.

In more detail, this algorithm works as follows: first eliminate all redundant RHS productions, until a fixed point is reached (none of the existing productions is not redundant with respect to members of the RHS). Then eliminate all redundant LHS productions by checking whether the beginner category (as specified by the input grammar) reaches each LHS. Again, removing rules with undefined LHS might cause rules with irrelevant RHS members. So after removing rules with redundant LHS, the procedure for eliminating redundant RHS members is called into action again. This entire process is repeated until a fixed point is reached.

6 Packing and Compression

This component carries out two tasks. First, it changes the internal structure of the grammar rules by packing rules that share LHSs together. This is a preparation to the next processing step in the pipeline, where all productions are printed in GSL format. Second, it compresses productions by looking for similar grammar rules that can be reduced to a single one.

Packing is in principle straightforward. Recall that the grammar is organized as a set of rules of the form cat(A,F,X_i) $\rightarrow \vec{C_i}$. The packing process changes this format so that the grammar is structured by a set of rules of the form cat(A,F) $\rightarrow \{\langle\vec{C_1},X_1\rangle,...,\langle\vec{C_n},X_n\rangle\}$, in such a way that there are no two occurrences of a LHS with different sets of RHSs in the grammar.

In the compression phase different productions with the same RHS are replaced by a single production. It works as follows:

1. Replace a pair of rules $C_i \rightarrow \mathcal{C}$ and $C_j \rightarrow \mathcal{C}$ $(i \neq j)$ by $C_k \rightarrow \mathcal{C}$ (where C_k is a new category);
2. Substitute C_k for all occurrences of C_i and C_j in the grammar.

Compression might be a potential factor to boost recognition speed because it reduces the number of derivations in the grammar (Dowding et al., 2001). Section 8 shows that this is indeed true.

7 Incorporating Semantics and Outputting GSL

The input of the last component of the compiler is a set of rules of the form LHS $\rightarrow \mathcal{C}$, where $\mathcal{C}$ is a set of ordered pairs of RHS categories and corresponding semantic values. The output of this component is a grammar in GSL format. First of all it writes the beginner node (as specified by the input grammar) with the `sem` slot. Suppose that the beginner category is S. Then this will yield:

```
.Grammar
[ S:a {<sem strcat($a ".")>} ]
```

Next, for each grammar rule, the LHS is converted to a single non-terminal GSL category symbol. Similarly, for each RHS in $\mathcal{C}$, all members are converted to (terminal or non-terminal) GSL categories. Some further book-keeping is required for associating the semantic representations in the UG rules with GSL slot-filling, because semantic values of the RHS in the UG rules are variables that have to be represented as lower-case atomic symbols in GSL, and the occurrences of these variables in the semantic representation of the LHS in GSL need to be prefixed with a \$, referring to the value of a slot (see Figure 2 for examples). This is carried out by maintaining a list of pairs of variables (occurring as semantic values for members of RHSs) and slot names (to be output in GSL format). This list is consulted when the semantic value of the LHS is output where the Nuance built-in `strcat/2` function is used for concatenating strings that describe the original semantics.

The final GSL rules are decorated with probabilities obtained from a (domain-specific) corpus. This is done by boot-strapping, where the corpus is parsed using a GSL grammar without probabilities, and the frequencies of rule applications are stored in a table. In a second compilation phase, this table is used to assign probabilities to each rule. Furthermore, a simple heuristic in case of structural ambiguities is used, where derivations with the closed attachments are preferred.

8 Evaluation and Comparison

The GSL compiler UNIANCE has been tested on a unification grammar (producing *underspecified discourse representation structures* as logical forms) for English containing 1238 lexical rules and 80 grammar rules, four of which suffered from left-recursion (one rule for verb phrase modification, and three rules for coordination of adjectives, nouns, and sentences). The performance of the different components during compilation of this grammar into GSL is presented in Table 1.

Table 1: Performance of the GSL compiler (Time measured in msecs).

Process	Rules	Time
UG productions	1318	2140
Feature instantiating	4444	591030
Eliminating left-recursion	5426	61040
Removing redundancies	5425	819330
Packing and Compression	272	343230
GSL productions	272	9020

Using version 7.0.4 of Nuance, the resulting GSL grammar, with 553 productions of which 252 where lexical entries[3], was compiled into a speech recognition package using the program `nuance-compile` with the options `-dont_flatten` (required for right-recursive grammars). The resulting language model (consisting of 1497 nodes) was tested on 131 utterances, comprising route instructions given to a mobile robot by 12 different native speakers (Lauria et al., 2001). To get an idea of the kind of data used, consider the transcript in Figure 4, consisting of four utterances (of the instructor).

er head to the end of the street – turn left – take the first left – er go right down the road past the first right and it's the next building on your right

Figure 4: Transcription of instruction `u8_GC_HD` taken from the IBL corpus.

[3]The 272 GSL rules are represented as disjunctions, as illustrated in Figure 2, but effectively they correspond to 553 productions.

Using Nuance's `batchrec` facility, this yielded an average recognition speed of 0.919xRT and a word error rate (WER) of 38.70% (of 1168 words). This relatively high average WER is due to the fact that around 44% of the utterances in the test suite are not covered by the unification grammar. Some of these utterances contain repairs or hesitations, some of them are non-grammatical, and others are due to their complications not covered by the current version of the grammar (mostly fragmentary input). Given this, the WER is actually not as bad as it first appears. (There are also speaker-dependent factors involved. Some speakers trigger very high WERs. For several speakers the WER drops down under 20%.)

Next, these results were compared with a GSL grammar that was produced without making use of the compression component (Section 6). This resulted in a GSL grammar with more rules (433 rather than 272), but the by NUANCE created language model had a similar amount of nodes (1488). However, this language model slightly slowed down the recognition process (1.017xRT) and gave a slightly higher average word error rate (39.13%). These results show the importance of this component in the GSL compiler and confirm findings presented in Dowding et al. (2001).

Finally, it was tested whether the addition of operations for computing semantic representations to GSL grammars affected the performance of speech recognition. The UNIANCE compiler was used to generate an identical grammar without semantic features. The average speech processing time for the grammar without semantic construction operations yielded a similar 0.921xRT (with the same WER of 38.70%). Hence, the experiment showed no substantial computational overhead in producing logical forms during speech recognition. Table 2 sums up the evaluation results.

Table 2: Evaluation of compiled GSL grammars tested on 131 utterances, using Nuance's `batchrec` facility. Sem=including semantic representations, Com=using rule compression.

Test	Rules	Nodes	Speed	WER
+Sem +Com	272	1497	0.919xRT	38.70
+Sem −Com	433	1488	1.017xRT	39.13
−Sem +Com	272	1497	0.921xRT	38.70

9 Conclusion and Future Work

It is possible to define a general method for encoding compositional semantics in language models for speech recognition. The method is based on compiling the semantic operations specified in a linguistic unification grammar into GSL, the grammar specification language used by Nuance's speech recognition software. The method is independent from the choice of logical form—as long as it includes the standard operations for compositional semantics (functional application and lambda abstraction). Furthermore, the method deals with transferring left-recursive grammar rules to right-recursive ones (a prerequisite for GSL, and also for other speech recognition grammars, including the JSpeech Grammar Format on which the W3C standard is based), while maintaining the compositional semantics stated in the original left-recursive production. The resulting speech recognition grammars show no loss of recognition speed, so the computational overhead in producing semantic representations is negligible (Table 2).

The GSL compiler, UNIANCE, is used as part of two projects involving speech recognition in human-machine dialogues. In the first project, IBL, users are engaged in a dialogue with a mobile robot, giving it directions how to reach a certain destination. The second project, D'Homme, investigates spoken dialogue in the home machine environment, where users are able to control domestic devices by spoken utterances. The use of UNIANCE has proven useful in these projects because (1) both scenarios have sufficiently sized domains for the GSL compiler to produce useful language models, and (2) semantic representations are directly produced by the speech recogniser and no separate processing step to build logical forms is needed.

Left recursion elimination for compiling speech grammars is discussed by Dowding et al. (2001), too. They report that the standard way of eliminating left recursion from a grammar caused problems to compile a working language model—contrary to the findings presented in this paper. They also mention an alternative transformation due to R. Moore, which they claim produces more compact left-recursion free grammars than the traditional algorithm. So it would be interesting to consider Moore's algorithm in future work on UNIANCE.

There are several ways to extend the GSL compiler. In terms of coverage, future work should address empty productions in the input UG, cases of non-immediate left-recursion and complex feature values. A further interesting option is to explore hybrid methods to language modelling, combining grammar-based and statistically-based approaches. This has been advocated by Knight et al. (2001) and is apparently available in recent versions of Nuance's speech recognition software.

Acknowledgements

This work was funded by the EU 5th Framework project D'Homme (Dialogues in the Home Environment) and IBL (Instruction-based Learning for Mobile Robots), via EPSRC grant GR/M90160.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1996. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley.

John Dowding, Beth Ann Hockey, Jean Mark Gawron, and Christopher Culy. 2001. Practical issues in compiling typed unification grammars for speech recognition. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, Toulouse, France.

Sylvia Knight, Genevieve Gorrell, Manny Rayner, David Milward, Rob Koeling, and Ian Lewin. 2001. Comparing grammar-based and robust approaches to speech understanding. In *Eurospeech 2001.*

Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, Johan Bos, and Ewan Klein. 2001. Training Personal Robots Using Natural Language Instruction. *IEEE Intelligent Systems*, pages 38–45, September/October.

Manny Rayner, Beth Ann Hockey, Frankie James, Elizbeth Owen Bratt, Sharon Goldwater, and Jean Mark Gawron. 2000. Compiling language models from a linguistically motivated unification grammar. In *The 18th International Conference on Computational Linguistics. Proceedings of the Conference.* Universität des Saarlandes, Saarbrücken, Germany.

Manny Rayner, John Dowding, and Beth Ann Hockey. 2001a. A baseline method for compiling typed unification grammars into context free language models. In *Eurospeech 2001*, pages 729–732.

Manny Rayner, Genevieve Gorrell, Beth Ann Hockey, John Dowding, and Johan Boye. 2001b. Do cfg based language models need agreement constraints? In *Proceedings of 2nd NAACL.*