Paper for *Advances in FDG* based on a Poster presented at ICFG12

Computational Representation of Underlying Structures and Lexical Entries using Domain-Specific Languages

Fabian Steeg, Christoph Benden, Paul O. Samuelsdorff[*]

July 15, 2007

This paper describes a modular system for generating sentences from formal definitions of underlying structures using domain-specific languages. The system uses Java as a general-purpose language, Prolog for lexical entries and the ANTLR parser generator for the definition and processing of custom domain-specific languages based on Funtional Grammar and Functional Discourse Grammar notation. It could be used in the context of a larger natural language processing system and as a tool for consistent formal notation in linguistic description.

Functional Grammar; Functional Discourse Grammar; Natural Language Processing; ANTLR; Domain-Specific Languages; Java; Prolog

1 Motivation and Overview

This paper describes a modular implementation of a system for generating sentences, using domain-specific languages (DSL; see section 3) for the formal representation of underlying structures and lexical entries. The DSL implemented for underlying structures is based on representations in Functional Grammar (FG; Dik 1997). Starting with a fully specified underlying structure instead of selecting lexical entries as the first step corresponds to the shift of Functional Discourse Grammar (FDG) to a top-down organization (Hengeveld & Mackenzie 2006, 668). Through its modular architecture the system can be extended for formal representations in FDG. Creating a computational implementation is a valuable evaluation tool for linguistic theories, since "linguistics may learn from being applied"

[*]Department of Linguistics, University of Cologne, Germany, http://www.uni-koeln.de/phil-fak/ifl, fsteeg@spinfo.uni-koeln.de, cbenden@spinfo.uni-koeln.de, paul-o.samuelsdorff@uni-koeln.de

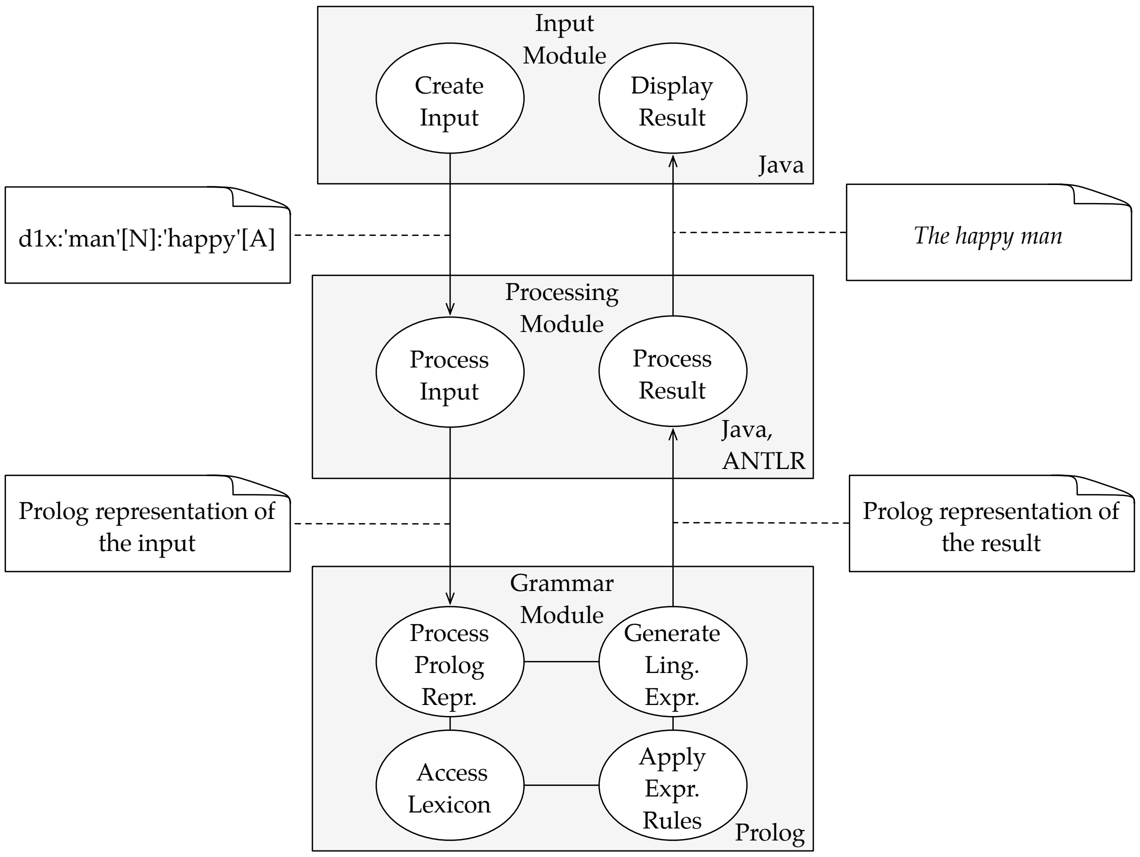
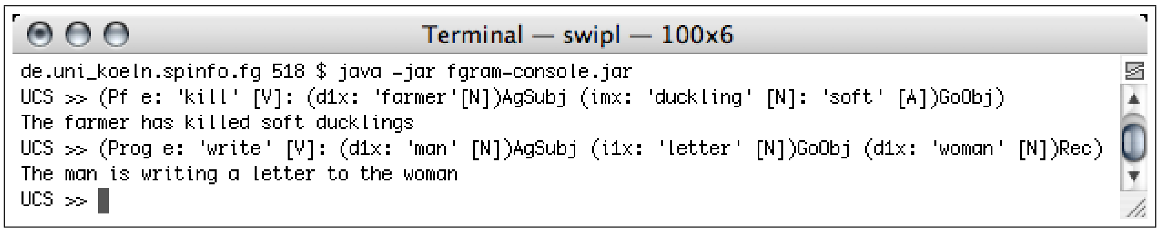


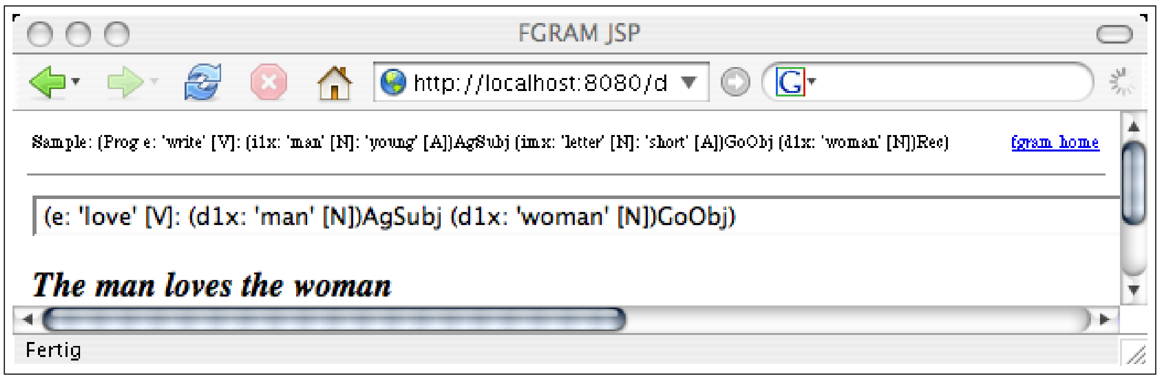
Figure 1: System Architecture

(Bakker 1994, 4). FDG explicitly demands “formal rigor” (Hengeveld & Mackenzie 2006, 668) and accordingly should be applied formally. By generating linguistic expressions from representations used in a theory, an implementation can be used to evaluate and improve the theory with respect to representational aspects and issues in language generation. The expression rules and the lexicon in our system are based on a revised and extended version of the implementation described in Samuelsdorff (1989). By means of its modular architecture the program could act as the language generation component in a larger natural language processing (NLP) system or as a tool for consistent formal notation in linguistic description.



```
de.uni_koeln.spinfo.fg 518 $ java -jar fgram-console.jar
UCS >> (Pf e: 'kill' [V]: (d1x: 'farmer'[N])AgSubj (imx: 'duckling' [N]: 'soft' [A])GoObj)
The farmer has killed soft ducklings
UCS >> (Prog e: 'write' [V]: (d1x: 'man' [N])AgSubj (i1x: 'letter' [N])GoObj (d1x: 'woman' [N])Rec)
The man is writing a letter to the woman
UCS >>
```

Figure 2: Screenshot of the console-based implementation



Sample: (Prog e: 'write' [V]: (i1x: 'man' [N]: 'young' [A])AgSubj (imx: 'letter' [N]: 'short' [A])GoObj (d1x: 'woman' [N])Rec)

(e: 'love' [V]: (d1x: 'man' [N])AgSubj (d1x: 'woman' [N])GoObj)

The man loves the woman

Figure 3: Screenshot of the web-based implementation

2 System Architecture

The system consists of individual, exchangeable modules for creating an underlying structure, processing that input and generating a linguistic expression from the input (cf. Fig. 1). In the *input module* an underlying structure is created, edited and evaluated. Upon evaluation the input is sent to the *processing module*, which communicates with the *grammar module*. When the generation is done, the user interface displays either the result of the evaluation, namely the linguistic expression generated from the input, or an error message. The system architecture can be characterized as a Model-View-Controller (MVC) or three-tier architecture. Such a modular approach has two main advantages: First, modules can be exchanged; for instance the *input module* is implemented both as a console application (cf. Fig. 2) and as a web-based user interface with the actual processing happening on a server (implemented using Java Server Pages on a Tomcat servlet container, cf. Fig. 3). Second, by using a defined input format, our system can be combined with other NLP components and be reused in new contexts.

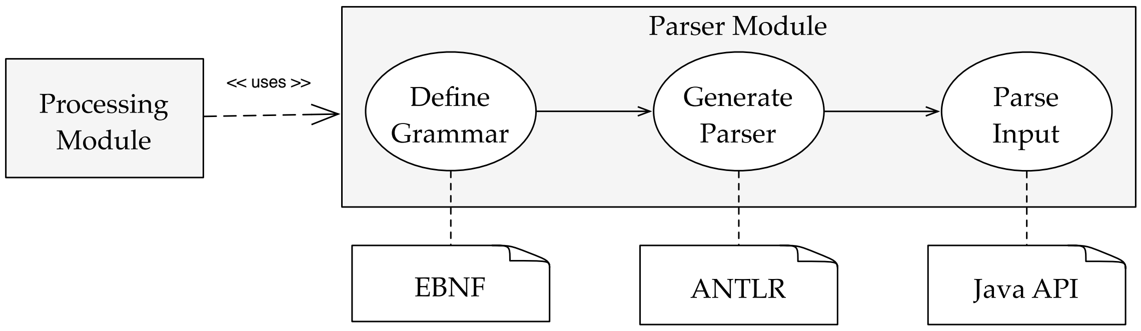


Figure 4: Parser overview

3 Domain-Specific Languages

The usage of languages which are tailored for a specific domain (domain-specific languages, DSL) has a long tradition in computing (e.g. for configuration files, like in the *sendmail* program, cf. Hunt & Thomas 1999, Ch. 12) and has been acknowledged as a best practice in recent years (Hunt & Thomas 1999, Ch. 12 and Parr 2007). Our system uses Java as a general-purpose language, Prolog as a DSL for lexical entries and expression rules, and a custom DSL for describing underlying structures, implemented using ANTLR, a tool for defining and processing domain-specific languages (Parr 2007). While e.g. in the domain of banking a DSL might describe credit rules, a linguist working with a model like FDG uses a DSL for linguistic description, e.g. for formal notation of underlying structures. With ANTLR, the form of the DSL is defined using a notation based on the *Extended Backus-Naur Form* (EBNF, cf. Fig. 7), from which a Java parser that can process the DSL is automatically generated, allowing for usage of the abundant supply of libraries available in Java (cf. Fig. 4 for an overview of the definition and the processing of the custom DSL).

4 Underlying Structures

The *processing module's* input format is a representation of the linguistic expression to be generated (cf. Fig. 5); its form is based on the representation of underlying structures given in Dik (1997). The *processing module* parses the input entered by the user and

```
(Past e:
    (d1x:'man'[N]:
        (Past Pf e:'give'[V]
            (d1x:'mary'[N])Ag
            (dmx:'book'[N]:'old'[A])Go
            (x:'man'[N])RecSubj
        )
    )
    (d1x:'john'[N])0
)
```

Figure 5: A nested underlying structure based on Dik (1997), which is parsable by the generated ANTLR v2 parser (represents *John is the man who was given the book by Mary*)

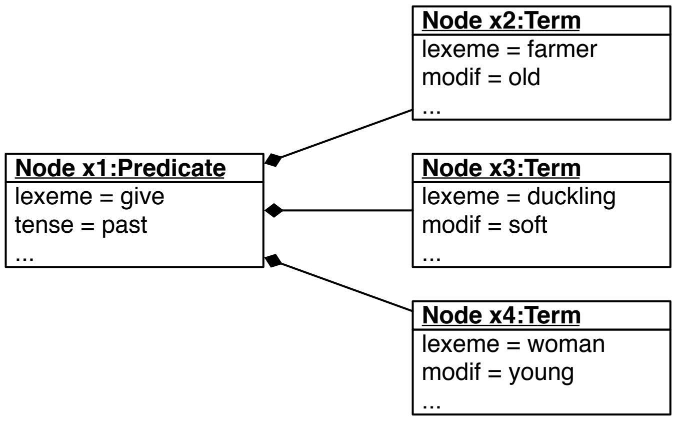


Figure 6: Internal representation of an underlying structure: a tree of Java objects (in UML notation)

creates an internal representation (cf. Fig. 6) which is then converted into the output format of the *processing module*, a Prolog representation of the input, which is used by the *grammar module* (cf. Ch. 5).

Figures 5 and 8 show the structural similarity of underlying structures in FG and FDG; both representations are nested parentheses, which can also be represented as trees (cf. Fig. 6). Since these representations can be described and processed in the same way, and representations on all levels of FDG have a common scheme (Hengeveld & Mackenzie 2006, 671), support for FDG representations is easy to add by supplying ANTLR definitions for representations on the individual levels in FDG, like the Interpersonal Level (IL) and the Representational Level (RL). Fig. 7 shows parser rules for structures on the RL,

```
pcontent   : '(' OPERATOR? 'p' X ( ':' head '(' 'p' X ')' )* ')' FUNCTION? ;
soaffairs  : '(' OPERATOR? 'e' X ( ':' head '(' 'e' X ')' )* ')' FUNCTION? ;
property   : '(' OPERATOR? 'f' X ( ':' head '(' 'f' X ')' )* ')' FUNCTION? ;
individual : '(' OPERATOR? 'x' X ( ':' head '(' 'x' X ')' )* ')' FUNCTION? ;
location   : '(' OPERATOR? 'l' X ( ':' head '(' 'l' X ')' )* ')' FUNCTION? ;
time       : '(' OPERATOR? 't' X ( ':' head '(' 't' X ')' )* ')' FUNCTION? ;
head       : LEMMA? ( '['
           ( soaffairs
           | property
           | individual
           | location
           | time )* ']' ) ? ;
```

Figure 7: ANTLR v3 parser rules for structures on the RL

```
(p1:[
    (Past e1:[
        (f1:tek[
            (x1:im(x1))Ag
            (x2:naif(x2))Inst
        ](f1))
        (f2:kot[
            (x1:im(x1))Ag
            (x3:mi(x3))Pat
        ](f2))
    ](e1))
](p1))
```

Figure 8: Underlying structure of a serial verb construction in Jamaican Creole (*Im tek naif kot mi*, 'He cut me with a knife', Patrick 2004, 290) on the RL in FDG, which is parsable by the parser generated from the rules in Fig. 7

from which a parser is generated that can parse expressions as the structure in Fig. 8, creating a representation which can be used for further processing (cf. Fig. 9). Such an ANTLR grammar definition provides a validator for the formal structure of IL and RL representations. Having an internal representation of the input, alternative processing to the creation of the corresponding linguistic expression is imaginable, like output of typeset representations of underlying structures in different formats, e.g. with or without indentation.

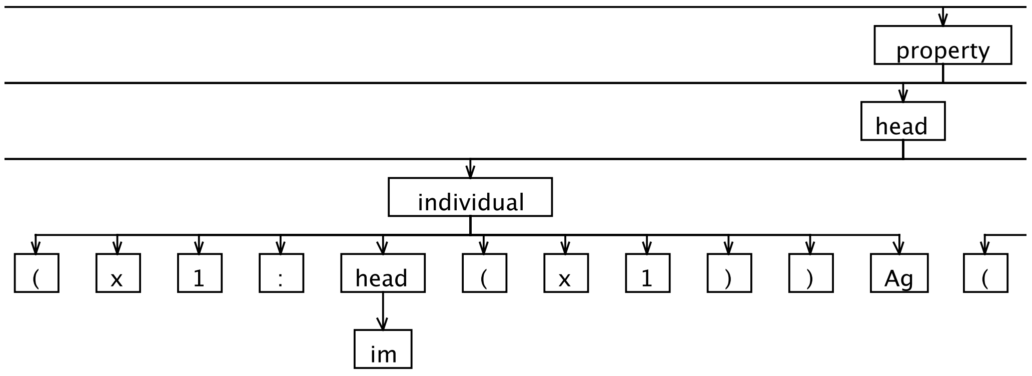


Figure 9: Part of the parse tree the generated parser produces for the expression in Fig. 8

5 Lexical Entries

In the *grammar module* the Prolog representation of the input generated by the *processing module* is used to generate a linguistic expression. Prolog was developed as a programming language for linguists and therefore offers convenient notation and processing mechanisms, e.g. lexical entries can be stored directly as Prolog facts (cf. Fig. 10). Prolog also has a particular strong standing as an implementation language for FG (e.g. Samuelsdorff 1989; Dik 1992). By restricting the usage of Prolog to lexical entries and expression rules and combining[1] it with other languages, instead of using it as a general-purpose programming language for the entire program, we use Prolog as a DSL in its original domain. The expression rules and the lexicon are based on a revised and extended version of the implementation described in Samuelsdorff (1989). To make the implementation work as a module in the described system, the user dialog of the original version (in which the underlying structure is built step by step) was replaced by an immediate processing of the entire input representing the linguistic expression to be generated. The user dialog is therefore replaced by the formal representation, which is created in the *input module* and converted into a Prolog representation by the *processing module*. This resembles the shift to a top-down organization (Hengeveld & Mackenzie 2006, 668) in FDG, where the conceptualization is the first step, not the selection of lexical elements, as it was in FG and in our original implementation.

[1]For calling Prolog from Java we use Interprolog (http://www.declarativa.com/interprolog/). The Prolog implementation we use is SWI-Prolog (http://www.swi-prolog.org/).

```
verb(
    give,
    action,
    [gave, given],
    [
        [agent, animate, X1],
        [goal, any, X2],
        [recipient, animate, X3]
    ],
    Sat
).
```

Figure 10: Ditransitive verb as a Prolog fact in the lexicon

6 Conclusion

We described a modular implementation[2] of a language generation system, representing underlying structures and lexical entries using domain-specific languages. The system makes use of an input format based on Dik (1997) and consists of modules implemented in Java, Prolog and ANTLR, making it easy to extend for FDG representations. As all structures used in FDG as well as the lexical entries (which are Prolog facts in our system) have a common tree structure, a unified implementation using ANTLR to define and process all these structures in the same manner as described for RL representations in Fig. 7, 8 and 9 is feasible and would allow further processing in all the target languages supported by ANTLR (currently Java, C, C++, C#, Objective-C, Python and Ruby). The system can be used to evaluate and improve FDG with respect to theoretical and representational issues; by means of its modular architecture it could act as the language generation component in a larger NLP system and as a tool for consistent formal notation in linguistic description.

[2]Infrastructure for collaborative development and the described implementation are available online under an open-source license (http://fgram.sourceforge.net).

References

Bakker, D.: 1994, *Formal and Computational Aspects of Functional Grammar and Language Typology*, Ph.D. thesis, Universiteit van Amsterdam.

Dik, S. C.: 1992, *Functional Grammar in Prolog: an Integrated Implementation for English, French and Dutch*, Mouton de Gruyter, Berlin, New York.

Dik, S. C.: 1997, *The Theory of Functional Grammar. Part 1: The Structure of the Clause (edited by Kees Hengeveld)*, second edn., Mouton de Gruyter, Berlin.

Hengeveld, K. & L. J. Mackenzie: 2006, 'Functional Discourse Grammar', in K. Brown (ed.), *Encyclopedia of Language and Linguistics*, second edn., Elsevier, Oxford, pp. 668–676.

Hunt, A. & D. Thomas: 1999, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional.

Parr, T.: 2007, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, The Pragmatic Bookshelf, Raleigh.

Patrick, P.: 2004, 'Jamaican Creole: Morphology and syntax.', in B. Kortmann, E. W. Schneider, C. Upton, R. Mesthrie & K. Burridge (eds.), *A Handbook of Varieties of English. Vol 2: Morphology and Syntax*, Topics in English Linguistics, Mouton de Gruyter, Berlin & New York, pp. 407–438.

Samuelsdorff, P. O.: 1989, 'Simulation of a Functional Grammar in Prolog', in J. H. Connolly & S. C. Dik (eds.), *Functional Grammar and the Computer*, De Gruyter, pp. 29–44.