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 10, 2007

This paper describes a modular system for generating sentences from formal definitions of underlying structures and lexical entries represented using domain-specific languages. The system is implemented using Prolog, Java and the ANTLR parser generator. It could be used in the context of a larger NLP system and as a tool for formal notation in linguistic description in FG and FDG.

FDG ; NLP ; Parsing ; ANTLR ; DSL ; 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). Through its modular architecture the system can be extended for formal representations in FDG. The idea of creating a computational implementation of FG and FDG mechanisms, to "build a model of the natural language user" (Dik 1997, 1) is central to these frameworks and a valuable evaluation tool for linguistic theories in general, since "linguistics may learn from being applied" (Bakker 1994, 4). Therefore our

*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

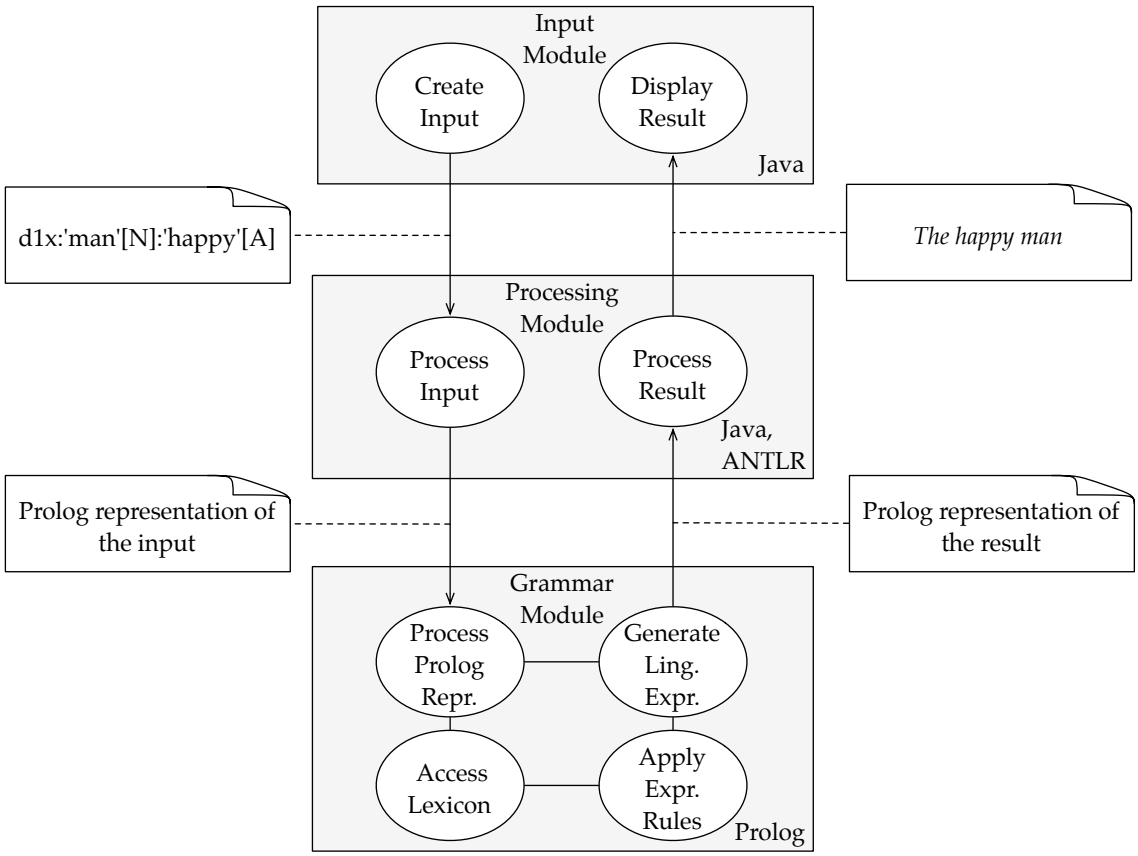

Figure 1: System Architecture

implementation can be used to evaluate and improve FDG with respect to theoretical issues in language generation. FDG demands "formal rigor" (Hengeveld & Mackenzie 2006, 668) and accordingly should be tested formally. Our implementation can therefore be used to evaluate and improve representational aspects of FDG. The expression rules and the lexicon 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 formal notation in linguistic description.

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 (see

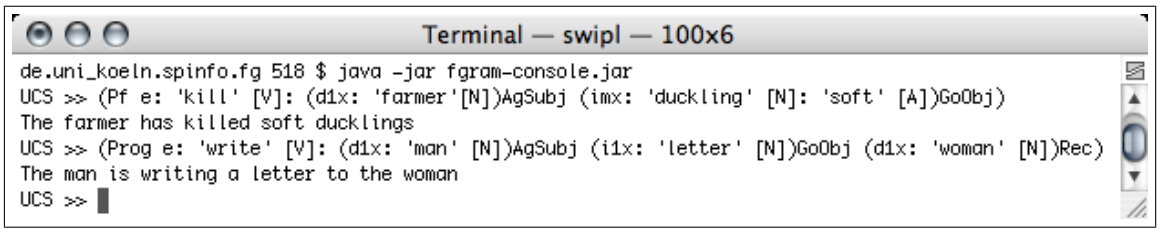

Figure 2: Screenshot of the console-based implementation

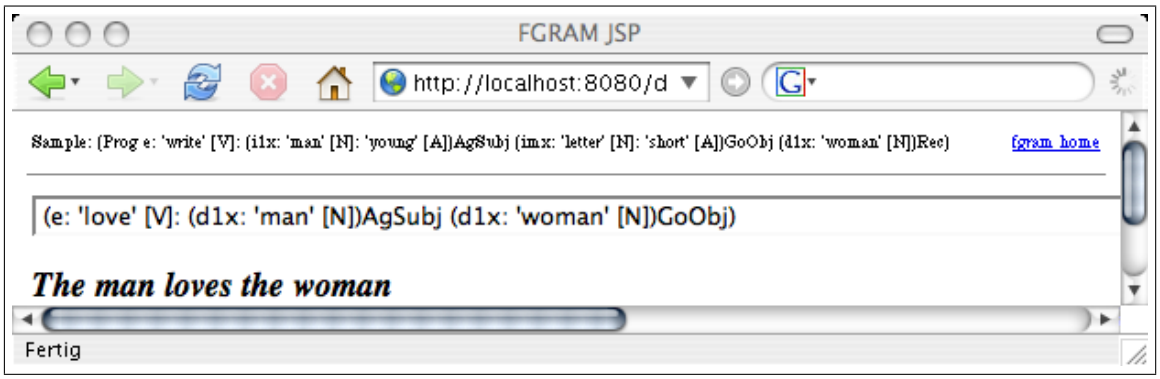

Figure 3: Screenshot of the Web-based implementation

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 graphical user interface (GUI) 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.

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 and has been acknowledged as a best practice in

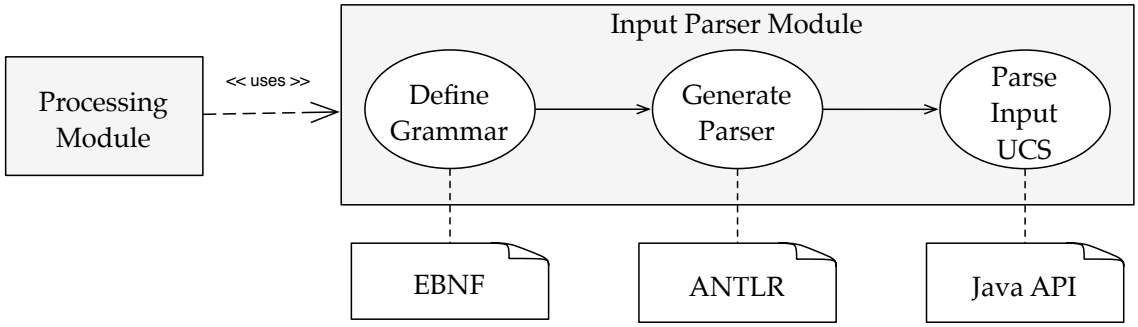

Figure 4: Parser overview

```
(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])O
)
```

Figure 5: A nested underlying structure based on Dik (1997), which is parsable by the generated ANTLRv2 parser (represents *John is the man who was given the book by Mary*)

recent years (cf. 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 self-defined 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 in EBNF notation (cf. Fig. 8), based on which a Java parser that can process the DSL is automatically generated, allowing for interaction with the abundant supply of libraries available in Java (cf. Fig. 4).

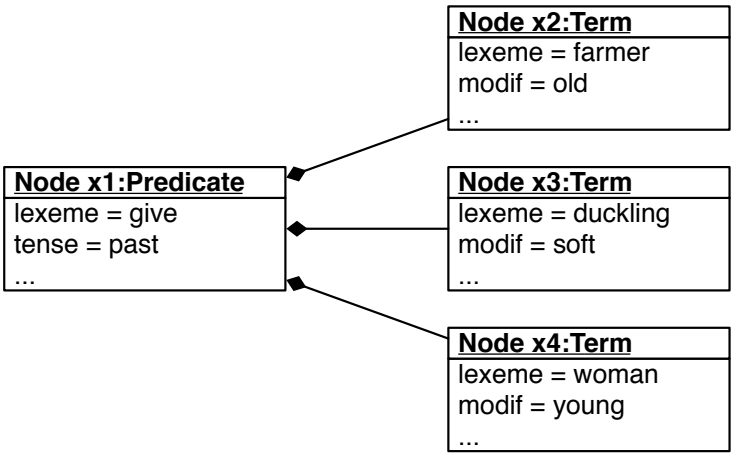

Figure 6: Internal representation of an underlying structure: a tree of Java objects (in UML notation)

4 Underlying Structures

The *processing module's* input format is a representation of the linguistic expression to be generated; its form is based on the representation of underlying structures given in Dik (1997). The *processing module* parses the input entered by the user (or potentially coming from a different source) and creates an internal representation (see 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*.

Figures 5 and 7 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 with the same mechanisms, 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, cf. Fig. 7, 8 and 9). Such an ANTLR grammar definition alone provides a validator for the formal structure of IL and RL representations. With grammar files for the IL and the RL implemented and having an internal representation of the input, alternative processing is possible too, e.g. output of typeset representations of the underlying structures with or without indentation.

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

```
content    : '(' OPERATOR? 'p' X ( ':' head '(' 'p' X ')' )* ')' FUNCTION? ;
soa        : '(' 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? ( '['
           ( soa
           | property
           | individual
           | location
           | time )* ']' ) ? ;
```

Figure 8: ANTLR v3 parser rules for structures on the RL

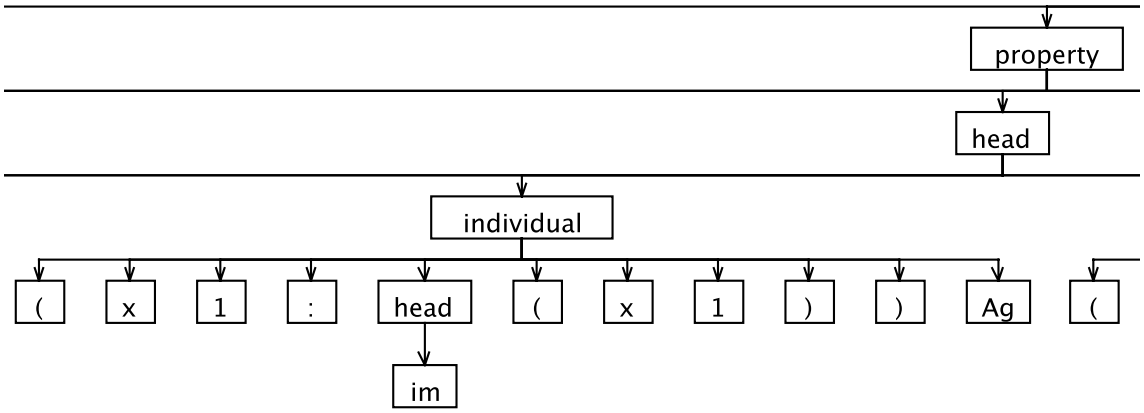

Figure 9: Part of the parse tree for the expression in Fig. 7

```
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

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 (see 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 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 in FDG (Hengeveld & Mackenzie 2006), where the conceptualization is the first step, not the selection of lexical elements, as it was in FG and in our original implementation.

6 Conclusion

We described a modular implementation 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 as Prolog facts have a common tree structure, a unified implementation using ANTLR to define and process all these structures is feasible and would allow further processing in all the target languages supported by ANTLR (i.e. Java, C, C++, 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 or as a tool for formal notation in linguistic description. The implementation and infrastructure for collaborative development is available at 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 Complete ANTLR Reference Guide: Building Domain-specific Languages*, The Pragmatic Bookshelf, Raleigh.

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.