

A modular system for generating linguistic expressions from underlying clause structures

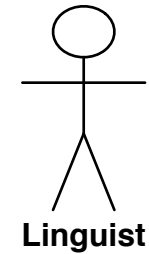
Using well-defined, FG-conformant notation

Motivation and Overview

The idea of creating a computational implementation of Functional Grammar (FG) mechanisms, to “build a model of the natural language user” (Dik 1997:1) is central to the theory of FG and a valuable evaluation tool for linguistic theories in general, since “linguistics may learn from being applied” (Bakker 1994:4).

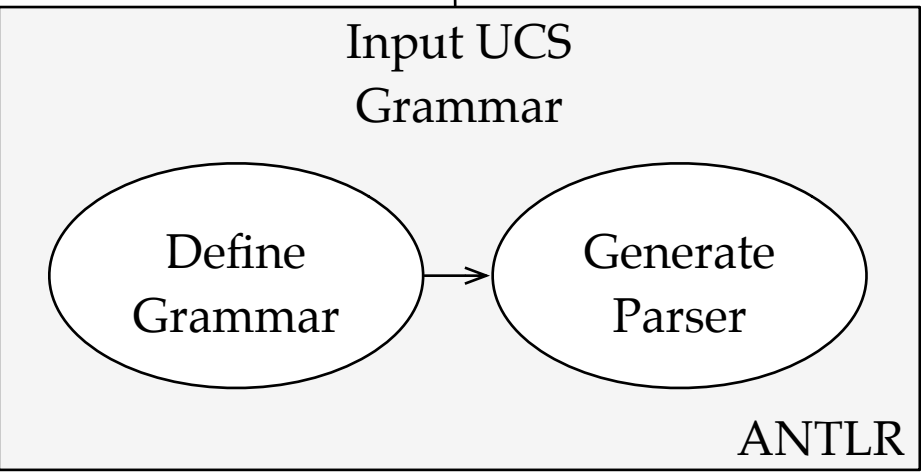
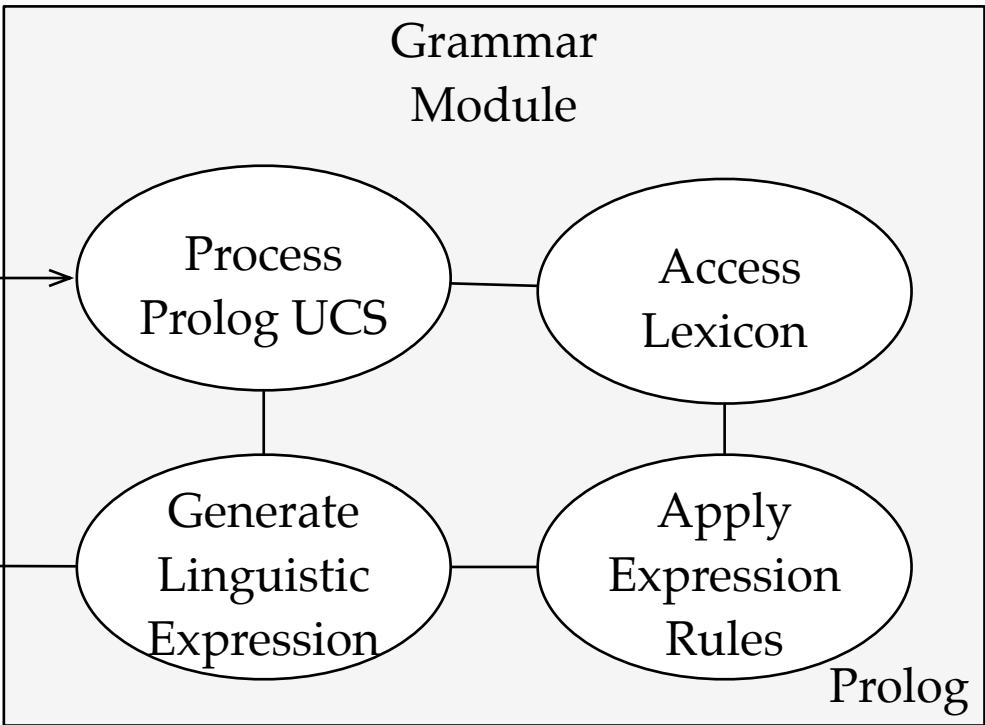
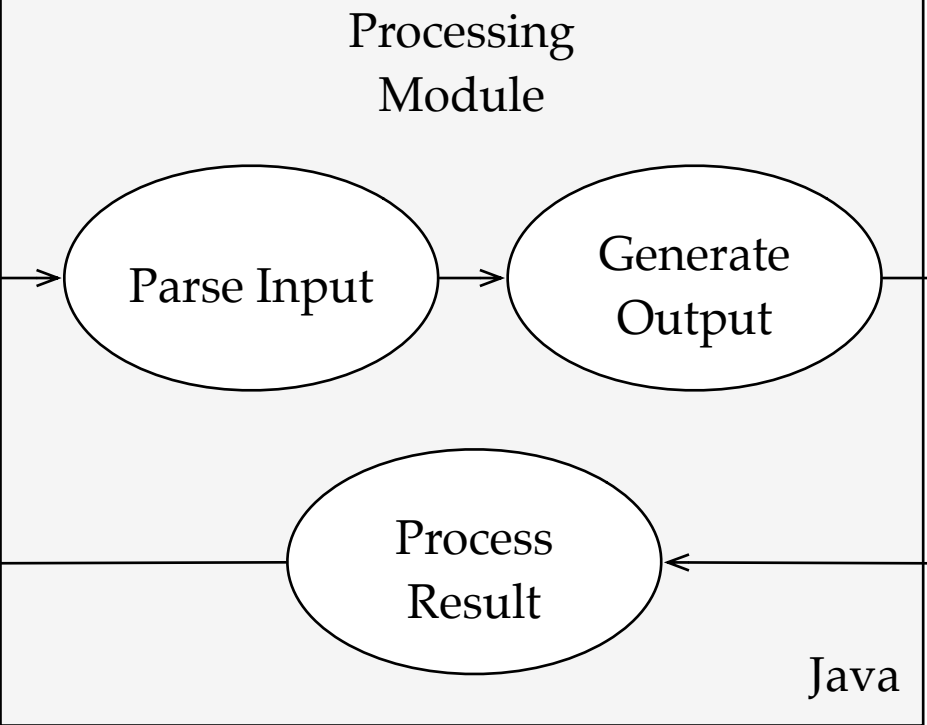
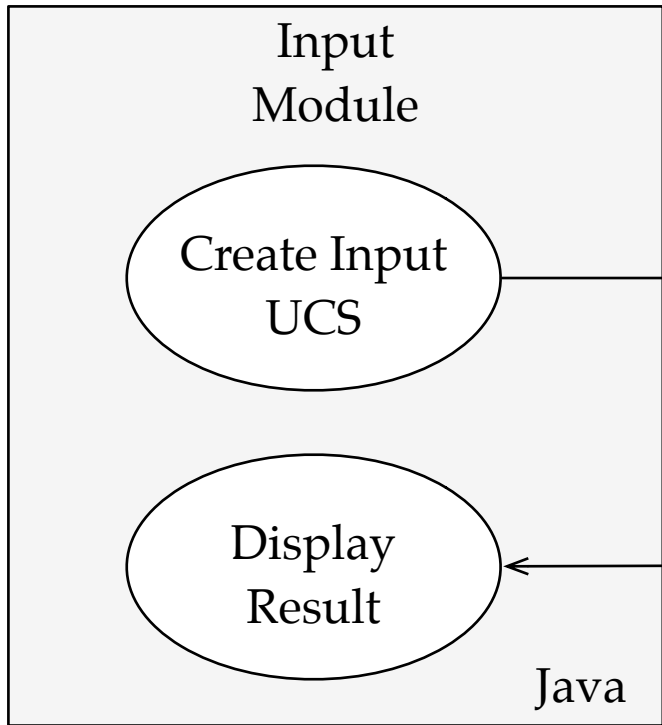
Therefore our system could be used to evaluate and improve the theory of FG with respect to theoretical issues in language generation. The system uses a UCS representation based on Dik (1997) and can therefore be used to experiment with representational issues of FG. The expression component is based on a revised version of the implementation described in Samuelsdorff (1989). By means of its modular architecture the system could act as the language generation component in a larger FG-based NLP system. The system is available for usage and participation at <http://fgram.sourceforge.net>.

In the original implementation the underlying structure is subsequently built via a user dialog, during which the expression to be generated is specified. To make the implementation work as a module in the described system, this user dialog is replaced by an immediate processing of the entire UCS representing the linguistic expression to be generated. The user dialog is therefore functionally replaced by the input UCS, which is created in the input module and converted into a Prolog representation by the processing module.



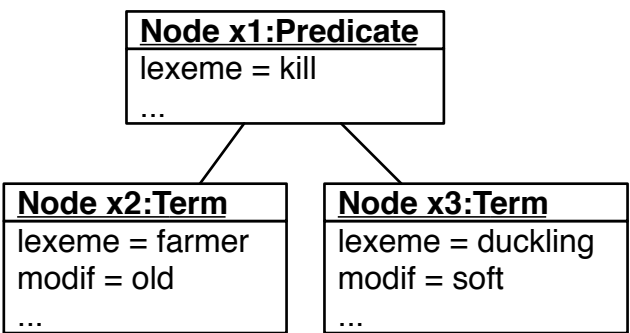
The old farmers had killed soft ducklings

(Past Pf e: 'kill' [V]: (dmx: 'farmer' [N]: 'old' [A])AgSubj (imx: 'duckling' [N]: 'soft' [A])GoObj)



Prolog representation of the result

Prolog representation of the input UCS



```
% PROLOG UCS

node(x1, 0).
node(x2, 1).
node(x3, 1).

prop(clause, illocution, decl).
prop(clause, type, mainclause).

prop(x1, type, pred).
prop(x1, tense, past).
prop(x1, perfect, true).
prop(x1, progressive, false).
prop(x1, mode, ind).
prop(x1, voice, active).
prop(x1, subnodes, [x2, x3]).
prop(x1, lex, 'kill').
prop(x1, nav, [V]).
prop(x1, det, def).

prop(x2, type, term).
prop(x2, role, agent).
prop(x2, relation, subject).
prop(x2, proper, false).
prop(x2, pragmatic, null).
prop(x2, num, plural).
prop(x2, modifs, [old]).
prop(x2, lex, 'farmer').
prop(x2, nav, [N]).
prop(x2, det, def).

prop(x3, type, term).
prop(x3, role, goal).
prop(x3, relation, object).
prop(x3, proper, false).
prop(x3, pragmatic, null).
prop(x3, num, plural).
prop(x3, modifs, [soft]).
prop(x3, lex, 'duckling').
prop(x3, nav, [N]).
prop(x3, det, indef).
```

```
// JAVA USAGE

InputProcessor processor = new InputProcessor(configFileLocation);
String ucs = "(e:'love'[V]:(x:'man'[N])AgSubj (dmx:'woman'[N])GoObj)";
String expression = processor.process(ucs, true);

// ANTLR GRAMMAR DEFINITION

[...]

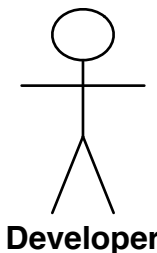
// "dix:" or "Impf Pf e:"
((d:DEF (n:NUMBER)? | ((tense:TENSE)? (aspect:ASPECT)?))? t:LAYER RESTRIKTOR
(w0:WORD p0:WORD_CLASS (RESTRIKTOR)? )?

[...]

LAYER : 'f' // predicate
      : 'x' // term
      : 'e' // predication
      : 'E' // clause
      : 'X' // proposition

NUMBER : '1' // sing
        : '2' // dual
        : 'm' // plural 'many'

[...]
```



Programming Languages

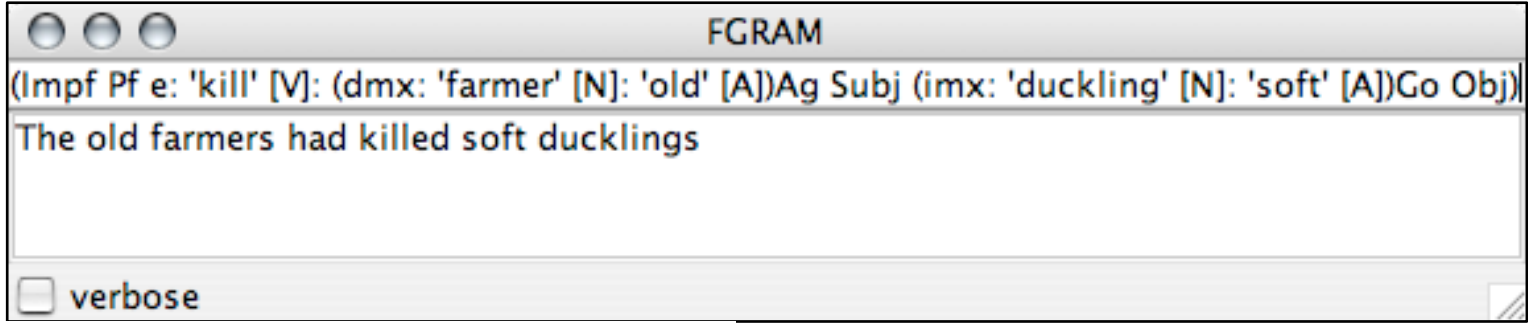
The System uses Java, Prolog and the ANTLR Grammar description language. The reason for using Java for the user interface and processing of the UCS, ANTLR for the Grammar definition and Prolog for the expression rules and the lexicon stems from the idea of using implementation languages well suited for a particular task. Java is a widespread multi-purpose programming language with abundant supply of libraries, ANTLR a specialized Grammar description language and Parser generator and Prolog offers convenient notation and processing mechanisms, is familiar to many linguists and has a particular strong standing as an implementation language for FG (e.g. Samuelsdorff 1989).

Modularity

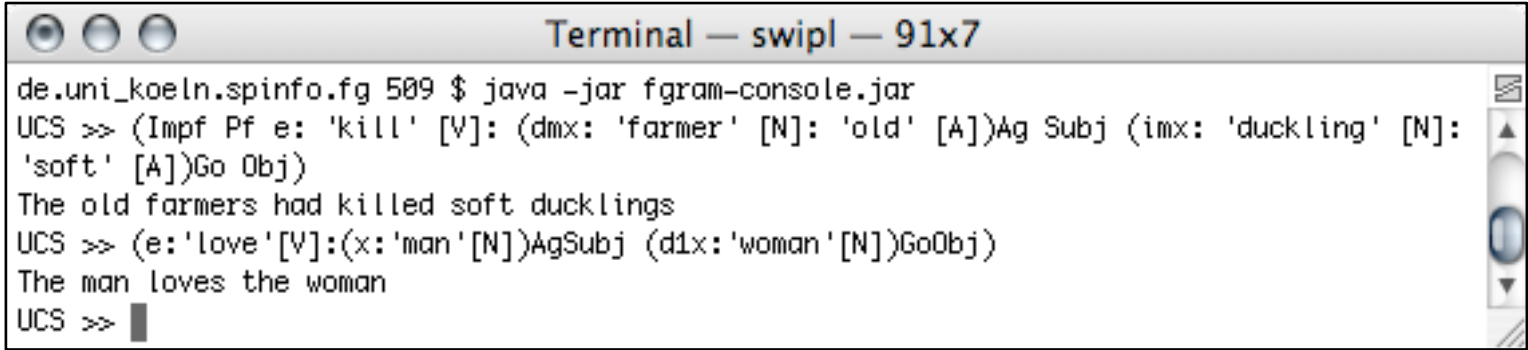
The system consists of individual, exchangeable modules for creating an underlying clause structure (UCS), processing that input and generating a linguistic expression from the input UCS. The system architecture can therefore be characterized as a Model-View-Controller (MVC) or Three-tier architecture. Such a modular approach has two main advantages: First, modules can be exchanged, for instance the input module could be a web-based user interface and the actual processing could happen on a server. Second, by using a defined input UCS format, our system could be combined with other FG-based NLP components which would have to formulate the input UCS for the processing module.

Different Implementations of the Input Module, using Java

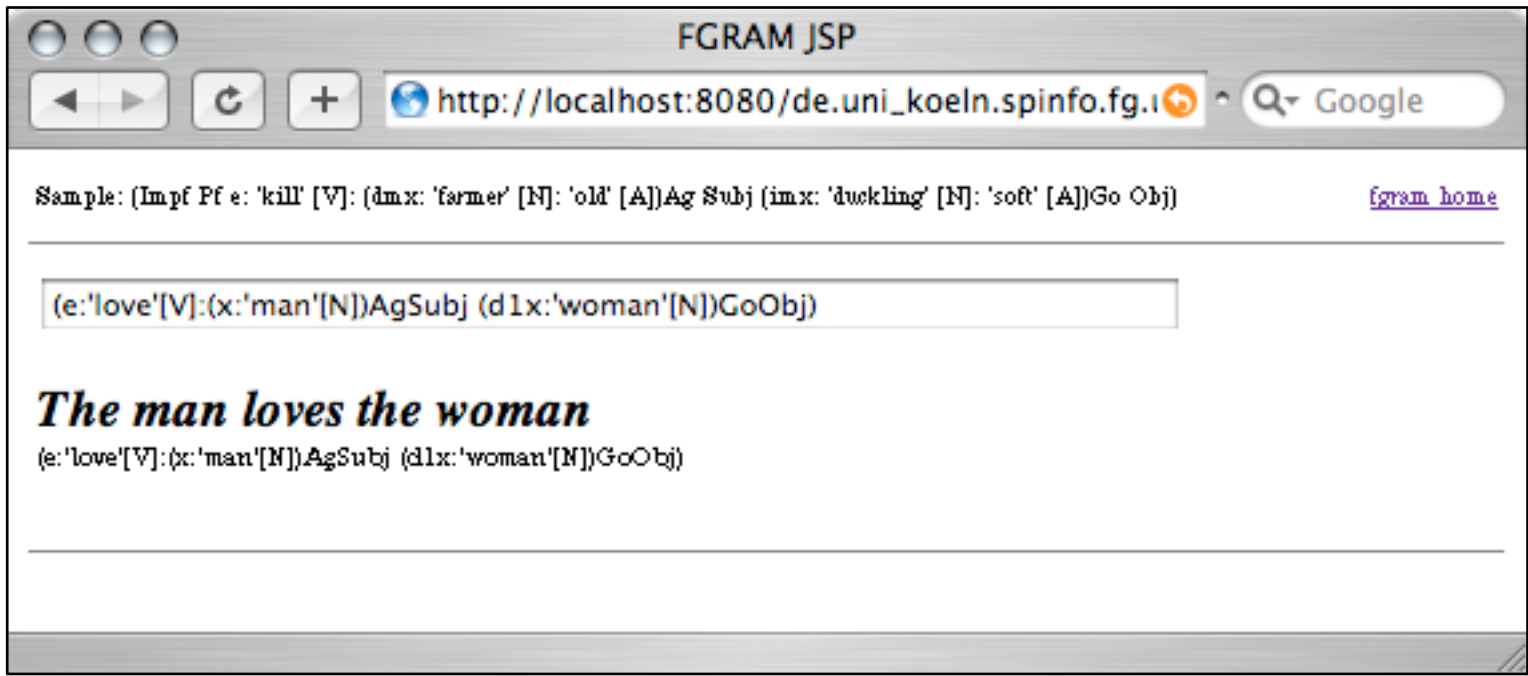
Any implementation in any language could act as the Input Module, if it fomulates (or assists in formulating) the input UCS for the Processing Module



Graphical User Interface using Java Swing



Java Console Application



Web-based Interface using Java Server Pages

References

BAKKER, Dik. 1994 *Formal and Computational Aspects of Functional Grammar and Language Typology*. Amsterdam: IFOTT.

DIK, Simon C. 1992 *Functional Grammar in Prolog; an integrated implementation for English, French and Dutch*. Berlin, New York: Mouton de Gruyter.

DIK, Simon C. 1997. *The Theory of Functional Grammar, Part 1: The Structure of the Clause*. Kees Hengeveld (ed.) 2nd rev. ed. Berlin, New York: Mouton de Gruyter.

SAMUELSDORFF, Paul O. 1989. Simulation of a Functional Grammar in Prolog. In: CONNOLLY, John H. & Simon C. DIK (eds.). *Functional Grammar and the Computer*. 29-44. Utrecht, Providence: Foris Publications.