Grammatical Framework (GF) for MT in sublanguage domains

Janna Khegai

Department of Computer Science,
Chalmers University of Technology,
SE-41296, Gothenburg, Sweden

janna@cs.chalmers.se

Abstract

Grammatical Framework (GF) is a meta-language for multilingual linguistic descriptions, which can be used to build rule-based interlingua MT applications in natural sublanguage domains. The GF open-source package contains linguistic and computational resources to facilitate language engineering including: a resource grammar library for ten languages, a user interface for multilingual authoring and a grammar development environment.

1 Introduction

Grammatical Framework (GF) is a grammar formalism that can be used to build rule-based interlingua MT applications in natural sublanguage domains (Burke & Johannisson, 2005; Bringert, Cooper, Ljunglöf, & Ranta, 2005; Caprotti, 2006). The GF implementation takes functional programming approach using a semantic model that could be described in Type Theory (Ranta, 2004). GF provides a powerful meta-language suitable for describing both natural and formal languages (Ljunglöf, 2004).

The core of a GF grammar is a language-independent interlingua, called **abstract syntax**. It models a domain by declaring categories and relations over them using the notation of functional programming languages. Abstract syntax is the most crucial and difficult part of grammar writing. Another part, called **concrete syntax**, maps abstract syntax into strings of natural language. Every language has its own definition for the given function, called **linearization**. Values returned by linearization could be not only strings, but also records and tables, see section 5.2 for some examples. This is especially important for expressing such language-specific features like inflections, morphological parameters and discontinuous constituents without affecting the abstract part.

Speaking of language-specific lower-level details we want to point out that it would be unreasonably tedious to descend to such details every time we write a GF grammar. To address the problem a standard library for the GF language, called **resource grammar** library, is provided. It decreases grammar development cost by code reuse, guaranteed grammaticality and raising the abstraction level of the task. Resource grammars are now implemented for ten languages: Danish, English, Finnish, French, German, Italian, Norwegian, Russian, Spanish and Swedish, see Fig. 1. They have been developed in parallel and share the same interface for common rules and categories, which makes implementation of both resource and application grammars easier (Ranta, to appear, 2005).

A resource grammar describes a language in general: the basic morphological and syntactical rules applicable to any domain. An **application grammar**, on the other hand, describes a particular sublanguage domain, for example, a set of math problems (Caprotti, 2006) or a local transport network (Bringert et al., 2005). To write an ap-

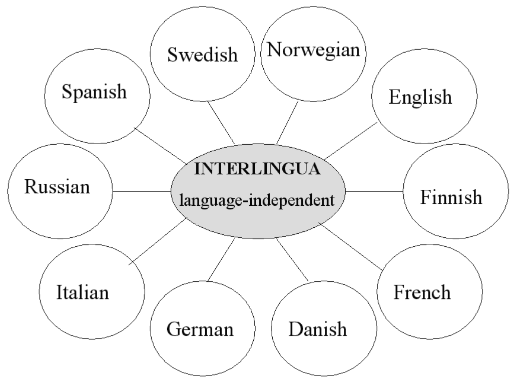


Figure 1: GF performs MT of interlingua type. Ten languages have general-purpose resource grammars that conform to a common interface.

plication grammar using the resource library one need to be a domain-expert, but from the linguistics point of view, it is enough to be a fluent speaker of a language. Thus, resource grammars take care of grammatical issues allowing the application grammarian to concentrate on the semantics of the described domain. Ideally, resource and application grammars should be separated on all possible levels, see Fig. 2. In our experience linguistic knowledge usually dominates the size of the grammar, which makes resource grammar library crucial for efficient grammar development.

Even with the resource library, writing grammars entirely by hand can be time-consuming because it still requires a fair knowledge of the resources. To speed up the process one can use Integrated Development Environment (GF IDE) (Khegai, 2005), included in the GF package – a grammar editor that can automatically suggest appropriate resource functions or even pre-fill the definition by parsing example strings with resource library. Manual post modification may be needed afterwards, but still it is useful if the system can at least partially fill-in the definition.

In the next sections we will outline how to write a demo application in GF. The source code and the executable are explained in sections 2 and 3 respectively. Section 4 contains notes on the expressiveness of the GF grammar formalism. Section 5 discusses some related work.

	Area	**Info**	**Grammarian**
Resource grammar	morphology, syntax	language-specific	linguist
Application grammar	semantics	language-independent	domain-expert

Figure 2: The table shows the ideal division of labor between resource and application grammars. A resource grammar is a general-purpose grammar that covers morphological and syntactical rules of a language. An application grammar is built on the top of the resource grammar and concentrates on the language-independent semantic description of a particular sublanguage domain.

2 Sample GF Grammar

We present a small example from the application grammar `Health` written using the resource grammar library in English, French, Swedish and Russian. We start by looking at the fragment of the language-independent (although the English names are used) abstract syntax:

```
cat
 Patient; Medicine; Prop;
fun
 ShePatient   : Patient;
 PainKiller   : Medicine;
 NeedMedicine : Patient ->
                Medicine -> Prop;
```

The categories `Patient`, `Medicine` and `Prop` denote a patient, a medication and a proposition respectively. `ShePatient` and `PainKiller` are constants of the types `Patient` and `Medicine`. The function `NeedMedicine` takes two arguments of the types `Patient` and `Medicine` and returns the result of the type `Prop` – a proposition expressing that a patient is in the need of a medication. `NeedMedicine` is used to form phrases like *she needs a painkiller* where patients and medications can vary. Thus, we get a generic function for forming this kind of propositions provided that a representative amount of possible arguments (various patients and medications) are covered by the grammar. The semantic tree (interlingua) of the phrase *she needs a painkiller* is a combination of the functions above:

```
NeedMedicine ShePatient PainKiller
```

where the constants `ShePatient` and `PainKiller` are used as arguments to the function `NeedMedicine`.

Given the abstract syntax now we need a corresponding concrete syntax in order to translate interlingua trees into strings of natural language. While the abstract syntax is shared among all the languages, each language has its own concrete syntax. Let us start with the linearization definitions, which happen to be the same for all five languages. This fragment is written using the language-independent part of the resource grammar library:

```
lincat
 Patient    = NP;
 Medicine   = NP;
 Prop       = S;
lin
 ShePatient = She;
```

The first three definitions indicate that `Patient`, `Medicine` and `Prop` categories will be expressed by noun phrase (`NP`) and sentence (`S`) categories. Noun phrases and sentences in different languages are already defined in the resource grammar, so we can just reuse them. The function `ShePatient` is basically a pronoun corresponding to the English pronoun *she*, which is also already defined in the resource grammar (`She`). Notice, that the function `She` bears the partial semantics of the pronoun *she*. Thus, some widely applicable semantic notions like pronoun references can be part of the resource grammar library, although, in general semantics is left for application grammars.

The definitions above are the same for all languages, which makes the porting trivial. In the remaining functions we see bigger differences:

```
-- English:
PainKiller =
 mkNP (nReg "painkiller");

-- French:
PainKiller =
 mkNP (nReg "calmant" masculine));
```

```
-- Swedish: PainKiller =
 mkNP (nIngenBöjning "smärtstillande");

-- Russian:
PainKiller =
  mkNP (nNeut_ee "обезболивающ");
```

`Painkiller` is defined by using the inflection paradigms `nReg` (pattern for `Reg`ular `n`ouns in English and French, see more details in section 5.2), `nIngenBöjning` (indeclinable nouns in Swedish) and `nNeut_ee` (neuter gender nouns ending with *-ee* in Russian) from the resource library, which take corresponding word stems (in quotes) as arguments. In French we also specify the gender (`masculine`) of a noun. The type-casting operation `mkNP` converts a noun into a noun phrase.

A linearization for `NeedMedicine` is defined as follows:

```
-- English:
NeedMedicine =
 predV2 (mkDirectVerb verbNeed);

-- Swedish: NeedMedicine =
 predV2 (mkDirectVerb verbBehöva);

-- French:
NeedMedicine patient medic =
 PredVP patient (avoirBesoin medic);

-- Russian:
NeedMedicine = predNeedAdjective;
```

The phrase *she needs a painkiller* is a transitive verb predication together with complementation in English and Swedish (`predV2`, see the function type signature below). In French the idiomatic expression *avoir besoin* (`avoirBesoin`) is used and, therefore, the more basic predication rule `PredVP` (the classic $NP\ VP \rightarrow S$ rule) is applied. Russian requires the rule for adjective predication (`predNeedAdjective`). All the functions are taken from the resource library. The function `mkDirectVerb` converts the lexicon entries `verbNeed` (English verb *to need*) and `verbBehöva` (Swedish verb *behöver*) into direct verb type. The arguments `patient` and `medic` of the function `NeedMedicine` denote a patient and a medication respectively.

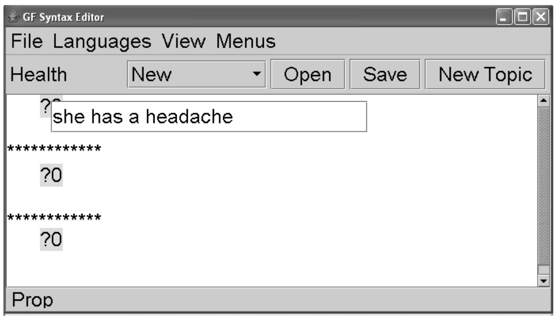
Notice, that to use, for example, the function `predV2` it is enough to know its type signature (implementation is hidden):

```
predV2 : TV -> NP -> NP -> S;
-- e.g. John loves Mary
```

The type signature indicates that `predV2` forms a sentence (`S`) combining a transitive verb (`TV`), a subject and an object (both expressed by noun phrases `NP`), like in *John loves Mary.* One just needs to recognize that the same pattern is used in the phrase *she needs a painkiller.* To define `NeedMedicine` we only have to supply the first verb argument – a transitive verb (in parenthesis in English and Swedish versions). The function `predV2` takes care of the rest including agreement, word order etc. We can even suppress both `NP` arguments in the notation, since they will be automatically restored from `predV2`'s type signature.

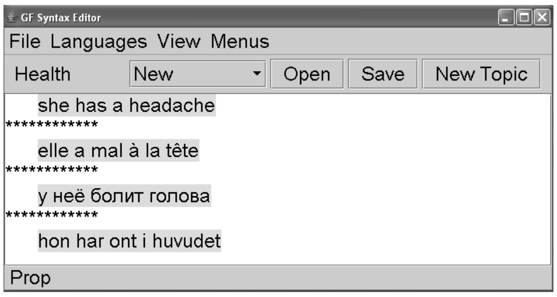
Having both abstract and concrete syntaxes for four languages we are now able to translate the sentence *she needs a painkiller* from one language into another via interlingua. In a similar manner we need to describe all utterances from the domain to be covered by the grammar. This requires a lot of work. The main part is abstract syntax – designing categories and functions to model the domain. All supported languages have to be taken into account in the interlingua representation, see section 3 (particulary Fig. 8) for an example. Sometimes, it is not possible to think of all the details from the start. Then, several iterations are needed along the way.

If a model conforms well to a language, writing a concrete syntax should be more or less straightforward using the GF IDE grammar editing tool. Its menu-driven mode helps to navigate through the resource library. Its example-based mode automatically pre-fills the linearization rules using parsing with resource grammars. For instance, to linearize `NeedMedicine` in English it is enough to provide an example like *you need vitamins* in GF IDE and the system will parse the string into a syntactic tree, which then can be modified into a linearization rule by replacing syntactic structures (e.g. *you* and *vitamins*) with corresponding



```
?0 she has a headache
************
?0
************
?0
```

Figure 3: GF syntax editor looks like a text-editor. Just type some text, for example, *she has a headache.*



```
she has a headache
************
elle a mal à la tête
************
у неё болит голова
************
hon har ont i huvudet
```

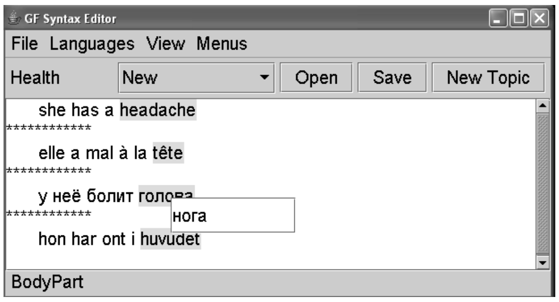
Figure 4: The sentence is translated via interlingua into French, Russian and Swedish.

semantic components (`patient` and `medic` arguments respectively).

3 Application example

We use the GF syntax editor (Khegai, Nordström, & Ranta, 2003) as a user interface to demonstrate the outcome of `Health` as a computer phrase-book, which is able to translate simple phrases on medical topics between four languages. One can start with typing something like *she has a headache*, see Fig. 3. The system parses the input into an interlingua representation, which is then linearized into strings in other languages, see Fig. 4.

One can proceed in any of the represented languages. For example, in Russian we can change the hurting body part from *голова* (*head*) to *нога* (*leg*), see Fig. 5. Of course, one cannot just type anything, since the system can only process a limited sublanguage. If in doubt, by right-clicking the mouse you can invoke context-dependent pop-up menu generated from the grammar, see Fig. 6. Notice, that the menu can be displayed not only in English, but also in all the other lan-


GF Syntax Editor

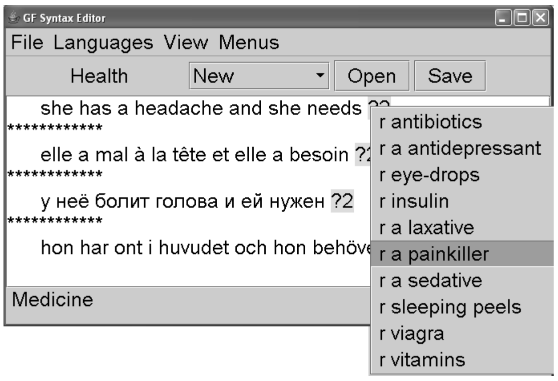
File Languages View Menus

Health New Open Save New Topic

she has a headache
****
elle a mal à la tête
****
у неё болит голова
****
нога
hon har ont i huvudet

BodyPart

Figure 5: Editing text in Russian: a middle-click on a chosen word pop-ups a text filed, which can be used for replacing the current body part – *голова* (*head*) by a new one – *нога* (*leg*).


GF Syntax Editor

File Languages View Menus

Health New Open Save

she has a headache and she needs ??
****
elle a mal à la tête et elle a besoin ?2
****
у неё болит голова и ей нужен ?2
****
hon har ont i huvudet och hon behöve

Medicine

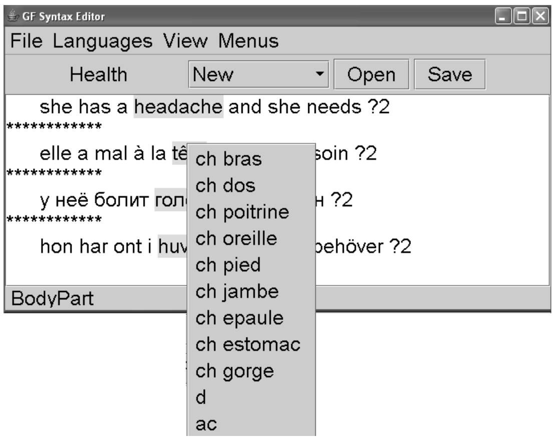
r antibiotics
r a antidepressant
r eye-drops
r insulin
r a laxative
r a painkiller
r a sedative
r sleeping peels
r viagra
r vitamins

Figure 6: An editing menu in English is invoked by right-clicking on a placeholder (denoted by a question mark) in the sentence in English. The menu is generated automatically from the grammar.

guages, for example, in French, see Fig. 7. So it is enough for the user to know only one of the languages.

Given a phrase in one language the system guarantees the correct translations into other languages. The translation is not only grammatically (agreement, word order etc.) but also stylistically correct. For example, *she has a headache* in English corresponds to *she has pain in the head* in Swedish and French, while in Russian it sounds more like *at her hurts head*. GF grammars allows us to enjoy high-quality translation by choosing the most appropriate form for the language, which, nonetheless, still conforms to the same underlying language-independent interlingua.

Interlingua approach has some inherent drawbacks. For example, the phrase *I have a headache* is considered ambiguous by the system, see Fig. 8. The reason is that the gender of the pronoun *I* used as a subject


GF Syntax Editor

File Languages View Menus

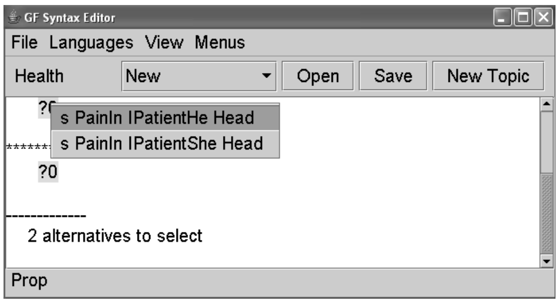
Health New Open Save

she has a headache and she needs ?2
****
elle a mal à la tê soin ?2
****
у неё болит гол н ?2
****
hon har ont i huv behöver ?2

BodyPart

ch bras
ch dos
ch poitrine
ch oreille
ch pied
ch jambe
ch epaule
ch estomac
ch gorge
d
ac

Figure 7: To get a context-dependent editing menu in French just right-click on the word in the French version.


GF Syntax Editor

File Languages View Menus

Health New Open Save New Topic

?0
s PainIn IPatientHe Head
s PainIn IPatientShe Head
******
?0

----
2 alternatives to select

Prop

Figure 8: In case of ambiguity the system asks to choose among the available options. Here, after typing *I have a headache* the gender of the sentence's subject is required. In English the gender of the subject is not important for forming a correct sentence. However, the gender distinction is kept in the interlingua semantic representation for the sake of compatibility with other languages where gender is needed for subject-verb agreement.

is not specified. The gender information is usually necessary for subject-verb agreement in, for example, Russian. So the system has to know the gender in order to potentially translate the statements into Russian. Notice, that Russian (or any other language) can be switched-off during the editing session, but it still affects the underlying semantic model.

4 On GF Expressiveness

The GF grammar formalism is stronger than context-free grammars. Parsing in GF consists of two steps:

- context-free parsing (a number of

parsers is implemented including basic top-down, Earley, chart)

- post-processing phase

The result produced by a context-free parser is further transformed by post-processing, which mainly consists of argument rearrangements and consistency checking for duplicated arguments. Consequently, a GF grammar needs to be translated into a context-free grammar, before feeding into a context-free parser. After such translation each GF rule is represented by a context-free rule annotated with so called **profile** that contains non-context-free information used by the post-processor. Profile describes the mapping from the position of a rule argument in the syntactic tree (after post-processing) to the position in the string (parsed text). Possible argument recombinations are:

- Permutation
- Suppression
- Reduplication

These operations are important for describing multilingual grammars sharing the same interlingua model (abstract syntax). For instance, permutation is used for translation of adjective modifiers from English into French: *even number* corresponds to *nombre pair*. Suppression is needed, for example, in translation from English into Russian, where the first language uses noun articles, but the second does not. In colloquial Russian reduplication of adjectives has an intensifying function like in *белый-белый снег* (*very white snow*). In some languages reduplication is used to form plural form (Lindström, 1995). The expressive power of the GF grammar formalism permits to handle these phenomena known to be non-context-free (Jurafsky & Martin, 2000).

To give an example of a profile annotation let us look at the GF function `f` for Finnish grammar that linearize strings like "*Every* woman is *pretty*":

```
fun
 f: A -> B -> C -> D;
 f x y z =
   y ++ "kuin" ++ y ++ "on" ++ z;
```

where `x`, `y` and `z` are the arguments of the type `A`, `B` and `C` respectively. We assume that all four types are linearized as strings. Function `f` corresponds to the context free rule:

```
 f ::= B "kuin" B "on" C
```

with profile:

[[], [1,2], [3]],

where each element in the list contains occurrences of the corresponding argument of the function. Positions are numbered according to the order in the right part of the resulting context-free rule. Thus, the first argument is suppressed, the second repeated twice on the first and second place in the rule. The third argument appears once at the third position.

Having at disposition the mechanisms for permutation, suppression and reduplication, we can easily describe the notorious non-context-free language:

$$\{a^n b^n c^n | n = 1, 2, ...\}$$

The corresponding GF grammar is the following:

```
cat
 S; Aux;

fun
 exp  : Aux -> S;
 first: Aux;
 next : Aux -> Aux;

lincat
  Aux = {s1:  Str; s2:  Str;
   s3:  Str};

lin
  exp x = {s = x.s1 ++ x.s2 ++ x.s3};
  first = {s1 = "a"; s2 = "b";
   s3 = "c"};
  next x = {s1 = "a" ++ x.s1; s2 =
   "b" ++ x.s2; s3 = "c" ++ x.s3};
```

The idea is to build an expression in two steps: first, accumulate each letter separately and second, glue the resulting strings together. For the first step we use an inductive definition parameterized by the variable `n`, namely: The function `first` forms a record containing just one of each letters *a*, *b* and *c*, describing the case when `n` equals one. The function `next` derives the `n+1`-case from the `n`-case. At the second step `exp` concatenates all the letters. `S` is a terminal string category, while `Aux` is an intermediate record category that contains three string fields – one for each letter. The syntax tree for *aaabbbccc* looks like:

```
exp (next(next first))
```

For a more systematic description of GF expressiveness and complexity we refer to (Ranta, 2004; Ljunglöf, 2004).

5 Related Work

GF is related to several well-established multilingual frameworks successfully used for MT applications such as Core Language Engine (CLE) (Rayner, Carter, Bouillon, Digalakis, & Wirén, 2000), Head-Driven Phrase Structure Grammar (HPSG) (Pollard & Sag, 1994) and Lexical-Functional Grammar (LFG) (Butt, King, no, & Segond, 1999). Unlike GF, which takes type-theoretical approach close to logical frameworks, they come from computational linguistics: feature-structured, unification-based and more focused on parsing.

5.1 Grammar Engineering Tools

The grammar engineering environments XLE (Xerox Linguistics Environment – LFG) (Crouch et al., 2005) and LKB (Lexical Knowledge Base – HPSG) (Copestake & Flickinger, 2000) have been used for building large scale multilingual grammars. Like LKB, GF is an open-source project, while XLE is not publicly available.

Both XLE and LKB have some Graphical User Interface (GUI), but mostly intended for running different commands from the command-line for processing the ready grammar files. Not much support is available for grammar writing itself. Grammars are written entirely by hand in an ordinary text editor like Emacs. GF IDE, on the other hand, is specially designed to meet the needs of grammar writers. The pluses comparing to common text editors are:

- Systematic treatment of "exotic" languages. UTF8 encoding is used for languages with non-latin alphabets. The system recognizes and properly displays non-latin characters automatically.
- Example-based, menu-driven grammar development.
- Lexicon extension on-the-fly, i.e. when an unknown word is encountered during example parsing the systems suggests to add the word to the resource lexicon and then repeats parsing attempt.

GF IDE saves time for scrolling the resource library files by hand and helps avoiding small syntactic mistakes and type-errors that can be automatically detected. It can also save efforts in learning the non-trivial grammar formalism, since otherwise substantial training is needed even for simple grammar writing.

5.2 CLE and GF Resource Grammar Library

GF resource grammar library is related to the proprietary CLE grammars used for Spoken Language Translator (SLT) system for Air Travel Information System (ATIS) domain. In the SLT system there are three main languages: English (coded first), Swedish and French (adapted from the English version). Spanish and Danish are also present in the CLE project.

Quasi (scope-neutral) Logical Form (QLF) – a feature-based formalism is used for representing language structures. Since the SLT uses a transfer approach two kinds of rules are needed:

- monolingual (to and from QLF-form) rules that are used for both parsing and generation.
- bilingual transfer rules.

Both sets are specified in (Rayner et al., 2000) using a unification grammars notation built on top of Prolog syntax (based on Definite Clause Grammars with features).

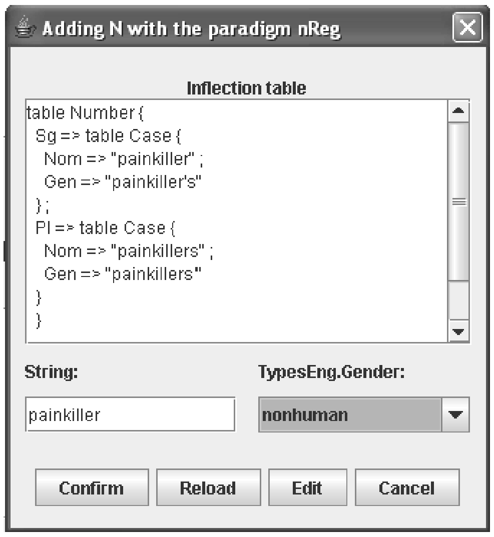
Both GF and CLE describe their grammars declaratively. Record fields in the GF type description roughly correspond to features in the CLE. Linearization (interlingua) rules in GF map to monolingual unification rules in CLE. However, no part of the GF is similar to the transfer rules set (more than one thousand rules for each language pair), since GF is essentially an interlingua system, although transfer components and statistical methods can be introduced.

The syntax coverage of the GF resource grammars is comparable with that of the CLE grammars (about one hundred rules per language in both cases), although, the same phenomena are not treated in the same way. For example, verb phrase discontinuous constituents are handled by combining the record fields in GF (in a manner similar to the one used in `exp`-rule in section 4), while there is a special set of ”movement” rules responsible for word order in the CLE. By having a structure inside a verb phrase, GF avoids introducing special rules for every word order, so the rules for forming verb phrases do not care about the word order in the final sentence. It is only on the very top sentence level, where the word order problem arises and is resolved by using the discontinuous constituents of a verb phrase.

Morphological rules in GF use tables while the corresponding CLE rules use features. In CLE we need to apply rules to the basic word form in order to get other forms. In GF the whole inflection pattern of a word (according to several parameters) is put in one table, see Fig 9. Therefore, we can just select a form from the table by specifying all the parameters at once, for example, to get the string *painkillers'*, we use the expression:

```
PainKiller.s !  Pl !  Gen
```

where `PainKiller` is the lexicon entry, the dot-operation gets access to the record field containing inflection table strings (`.s`), the exclamation-sign-operation (`!`) selects the corresponding form (plural, genitive) from



```
table Number {
 Sg => table Case {
  Nom => "painkiller" ;
  Gen => "painkiller's"
 } ;
 Pl => table Case {
  Nom => "painkillers" ;
  Gen => "painkillers"
 }
}
```

String: painkiller

TypesEng.Gender: nonhuman

Figure 9: The GF IDE dialog window for adding lexicon entries. As indicated by the window caption, the inflection table has been generated from the stem string `painkiller` by using the `nReg` (`Reg`ular `n`oun) inflection pattern. We can see that all declension forms (by number: `Sg`, `Pl` and case: `Nom`,`Gen`) are kept together in one table. The inherent parameter `Gender` is also kept as a record field in the noun category.

the table. Several independent rules are needed to express a similar pattern in CLE, since one rule can only take care of one parameter at a time. A possible explanation for such differences in lexicon construction is that CLE is more parsing-oriented, so keeping all the forms in one entity is not crucial, while GF is more generation-oriented and storing all the forms together is more convenient during generation, especially for languages with rich inflectional systems.

Thus, the differences are partly due to design decisions, partly hereditary to formalisms' expressive means. However, the general structure of the GF resource library and the CLE monolingual rule set match a lot, which is only natural, since they both reflect the structure of the modelled language.

5.3 Multilingual Authoring

The GF syntax editor from section 3 originates from proof editors like Alf (Magnusson & Nordström, 1994) used for interactive theorem proving and pretty-printing of the proofs. Constructing a proof in a proof editor corresponds to constructing an abstract syntax tree in GF. The concrete part is, however, missing from the proof editors, since the proofs are usually expressed in a symbolic language of mathematics.

Menu-driven multilingual authoring procedure is similar to the WYSIWYM tool (Power, Scott, & Evans, 1998), where Multilingual Natural Language Generation from a semantic knowledge base expressed in a formal language (non-linguistic source) is opposed to MT (linguistic source). However, there are two important differences. First, GF grammars are not hard-wired and can be extended and changed. This makes GF more generic compared to WYSIWYM. Second, GF grammar is bidirectional, so for every grammar not only the generator is produced, but also a parser. Thus, the author is allowed to type his input provided that it conforms to the grammar, which is useful for multilingual authoring applications because typing can speed up tedious menu editing. GF syntax editor is also capable of handling ambiguous input.

The language-independent ontology (domain model, terminology) in the WYSIWYM corresponds to abstract syntax in GF. Respectively, building a knowledge diagram in the WYSIWYM corresponds to the construction of an abstract syntax tree in GF. In both systems a feedback text, generated from the current object in several languages (English, French and Italian for WYSIWYM) is shown to the user while editing.

Even the architecture of the WYSISYM implementations DRAFTER-II is similar to GF in a way that the GUI part is separated from the processing engine. In WYSIWYM, Prolog is used for both ontology description and generation while GUI is written in CLIM (Common Lisp Interface Manager). In GF, the computational core is written in a functional programming language Haskell, while GUI is a Java program.

GF was one of the sources of inspiration for an XML-based multilingual document authoring application for pharmaceutical domain developed at Xerox Research Center Europe (XRCE) (Dymetman, Lux, & Ranta, 2000). Its grammar formalism called Interaction Grammars (IG) also has a separation between the language-independent interlingua (abstract syntax in GF) and parallel realization grammars (concrete syntax in GF) for different languages (English and French). As GF the IG also uses the notions of typing and dependent types and is suitable for both parsing and generation. But unlike GF the IG comes from the logic programming tradition. Like CLE grammars (see subsection 5.2) it is based on the Definite Clause Grammars – a unification-based extension of context-free grammars, which has a build-in implementation in Prolog.

6 Conclusion

GF is an open-source platform for building rule-based MT applications of interlingua type. It has two-level organization: abstract syntax for semantic definitions (interlingua) projected onto the concrete syntaxes in every supported language. The division between abstract and concrete syntax allows grammar writers to focus on the semantic level, abstracting from the structural differences between languages.

The division between general-purpose resource grammars and domain-specific application grammars allows for mapping interlingua into surface syntactic representations without descending to low-level language-specific linguistic details. The mapping can be even performed semi-automatically using example-based menu-driven grammar development interface (GF IDE).

Designed for generation rather than parsing, GF works best for well-formalized sublanguage domains like software specifications (Burke & Johannisson, 2005), mathematical language (Caprotti, 2006) or transport networks (Bringert et al., 2005). The end-user applications so far comprise multilingual authoring tools (Hähnle, Johannisson, & Ranta, 2002; Caprotti, 2006) and multimodal dialog systems (Cooper & Ranta, 2004; Bringert et al., 2005). The GF language processor including several grammar engineering tools is available at GF's homepage (Ranta, 2006).

References

Bringert, B., Cooper, R., Ljunglöf, P., & Ranta, A. (2005). Multimodal Dialogue System Grammars. In *DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France.*

Burke, D., & Johannisson, K. (2005). Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In J. B. P. Blace, E. Stabler & R. Moot (Eds.), *Logical Aspects of Computational Linguistics (LACL 2005)* (Vol. 3402, pp. 51–66). Springer.

Butt, M., King, T. H., no, M.-E. N., & Segond, F. (Eds.). (1999). *A grammar writer's cookbook.* Stanford: CSLI Publications.

Caprotti, O. (2006). WebALT! Deliver Mathematics Everywhere. In *SITE 2006, Orlando, USA.* (`webalt.math.helsinki.fi/content/e16/e301/e512/PosterDemoWebALT.pdf`)

Cooper, R., & Ranta, A. (2004). Dialogue systems as proof editors. *The Jornal of Logic, Language and Information.*

Copestake, A., & Flickinger, D. (2000). An open-source grammar development environment and broad-coverage english grammar using hpsg. In *Second conference on Language Resources and Evaluation (LREC-2000), Athens, Greece.*

Crouch, D., Dalrymple, M., Kaplan, R., King, T., Maxwell, J., & Newman, P. (2005). *XLE documentation.* (URL: `www2.parc.com/istl/groups/nltt/xle`)

Dymetman, M., Lux, V., & Ranta, A. (2000). XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany* (pp. 243–249).

Hähnle, R., Johannisson, K., & Ranta, A. (2002). An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche & H. Weber (Eds.), *Fundamental Approaches to Software Engineering* (Vol. 2306, pp. 233–248). Springer.

Jurafsky, D., & Martin, J. (2000). *Speech and language processing.* Prentice Hall.

Khegai, J. (2005). *GF IDE for GF 2.1.* `www.cs.chalmers.se/~aarne/GF2.0/GF-Doc/GF_IDE_manual/index.htm`.

Khegai, J., Nordström, B., & Ranta, A. (2003). Multilingual syntax editing in GF. In A. Gelbukh (Ed.), *CICLing-2003, Mexico City, Mexico* (pp. 453–464). Springer.

Lindström, J. (1995). *Summary on reduplication.* LINGUIST List: Vol-6-52.

Ljunglöf, P. (2004). *Expressivity and Complexity of the Grammatical Framework.* (URL: `www.cs.chalmers.se/~peb/pubs/p04-PhD-thesis.pdf`)

Magnusson, L., & Nordström, B. (1994). The ALF proof editor and its proof engine. In *Types for Proofs and Programs* (pp. 213–237). Springer.

Pollard, C., & Sag, I. (1994). *Head-Driven Phrase Structure Grammar.* University of Chicago Press.

Power, R., Scott, D., & Evans, R. (1998). Generation as a solution to its own problem. In *Inlg'98.* Niagara-on-the-Lake, Canada.

Ranta, A. (2004). Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming, 14*(2), 145–189.

Ranta, A. (2006). *GF Homepage.* (`www.cs.chalmers.se/~aarne/GF/`)

Ranta, A. (to appear, 2005). Modular Grammar Engineering in GF. *Research in Language and Computation.* (URL: `www.cs.chalmers.se/~aarne/articles/ar-multieng.pdf`)

Rayner, M., Carter, D., Bouillon, P., Digalakis, V., & Wirén, M. (2000). *The spoken language translator.* Cambridge University Press.