GF Parallel Resource Grammars and Russian

Janna Khegai
Department of Computer Science
Chalmers University of Technology
SE-41296 Gothenburg, Sweden
janna@cs.chalmers.se

Abstract

A resource grammar is a standard library for the GF grammar formalism. It raises the abstraction level of writing domain-specific grammars by taking care of the general grammatical rules of a language. GF resource grammars have been built in parallel for eleven languages and share a common interface, which simplifies multilingual applications. We reflect on our experience with the Russian resource grammar trying to answer the questions: how well Russian fits into the common interface and where the line between language-independent and language-specific should be drawn.

1 Introduction

Grammatical Framework (GF) (Ranta, 2004) is a grammar formalism designed in particular to serve as an interlingua platform for natural language applications in sublanguage domains. A domain can be described using the GF grammar formalism and then processed by GF. Such descriptions are called **application grammars**.

A **resource grammar** (Ranta, to appear) is a general-purpose grammar that forms a basis for application grammars. Resource grammars have so far been implemented for eleven languages in parallel. The structural division into **abstract** and **concrete** descriptions, advocated in GF, is used to separate the language-independent common interface or **Application Programming Interface** (**API**) from corresponding language-specific implementations. Consulting the abstract part is sufficient for writing an application grammar without descending to implementation details. This approach raises the level of application grammar development and supports multilinguality, thus, providing both linguistic and computational advantages.

The current coverage is comparable with the Core Language Engine (CLE) project (Rayner et al., 2000). Other well-known multilingual general-purpose grammar projects that GF can be related to, are LFG grammars (Butt et al., 1999) and HPSG grammars (Pollard and Sag, 1994), although their parsing-oriented unification-based formalisms are very different from the GF generation-oriented type-theoretical formalism (Ranta, 2004).

A Russian resource grammar was added after similar grammars for English, Swedish, French and German (Arabic, Italian, Finnish, Norwegian, Danish and Spanish are also supported in GF). A language-independent API representing the coverage of the resource library, therefore, was already available. The task was to localize modules for Russian.

A resource grammar has morphological and syntactic modules. Morphological modules include a description of word classes, inflectional paradigms and a lexicon. Syntactic modules comprise a description of phrasal structures for analyzing bigger than one-word entities and various combination rules. Note, that semantics, defining the meanings of words and syntactic structures, is constructed in application grammars. This is because semantics is rather domain-specific, and, thus, it is much easier to construct a language-independent semantic model for a particular domain than a general-purpose resource semantics.

In the following sections we consider typical definitions from different resource modules focusing on aspects specific to Russian. We will also demonstrate the library usage in a sample application grammar.

2 Word Classes

Every resource grammar starts with a description of word classes. Their names belong to the language-independent API, although their implementations are language-specific. Russian fits quite well into the common API here, since like all other languages it has nouns, verbs, adjectives etc. The type system for word classes of a language is the most stable part of the resource grammar library, since it follows traditional linguistic descriptions (Shelyakin, 2000; Wade, 2000; Starostin, 2005). For example, let us look at the implementation of the Russian adjective type `AdjDegree`:

```
param
 Degree  = Pos | Comp | Super;
 Case = Nom|Gen|Dat|Acc|Inst|Prep;
 Animacy = Animate | Inanimate;
 Gender = Masc | Fem | Neut;
 GenNum = ASingular Gender|APlural;
 AdjForm = AF Case Animacy GenNum;

oper
 AdjDegree : Type =
  {s : Degree => AdjForm => Str};
```

First, we need to specify parameters (`param`) on which inflection forms depend. A vertical slash (`|`) separates different parameter values. While in English the only parameter would be comparison degree (`Degree`), in Russian we have many more parameters:

- `Case`, for example: *большие дома – больших домов* (*big houses – big houses'*).
- `Animacy` only plays a role in the accusative case (`Acc`) in masculine (`Masc`) singular (`ASingular`) and in plural forms (`APlural`), namely, accusative animate form is the same as genitive (`Gen`) form, while accusative inanimate form is the same as nominative (`Nom`): *Я люблю большие дома – я люблю больших мужчин* (*I love big houses – I love big men*).
- `Gender` only plays role in singular: *большой дом – большая машина* (*big house – big car*). The plural never makes a gender distinction, thus, `Gender` and number are combined in the `GenNum` parameter to reduce redundant inflection table items. The possible values of `GenNum` are `ASingular Masc`, `ASingular Fem`, `ASingular Neut` and `APlural`.
- `Number`, for instance: *большой дом – большие дома* (*a big house – big houses*).
- `Degree` can be more complex, since most Russian adjectives have two comparative (`Comp`) forms: declinable attributive and indeclinable predicative[1]: *более высокий* (*more high*) – *выше* (*higher*), and more than one superlative (`Super`) forms: *самый высокий* (*the most high*) – *наивысший* (*the highest*).

Even another parameter can be added, since Russian adjectives in the positive (`Pos`) degree have long and short forms: *спокойная река* (*the calm river*) – *река – спокойна* (*the river is calm*). The short form has no case declension, thus, it can be considered as an additional case (Starostin, 2005). Note, that although the predicative usage of the long form is perfectly grammatical, it can have a slightly different meaning compared to the short form. For example: long, predicative *он – больной* (*"he is crazy"*) vs. short, predicative *он – болен* (*"he is ill"*).

An `oper` judgement combines the name of the defined operation, its type, and an expression defining it. The type for degree adjective (`AdjDegree`) is a table of strings (`s: .. => ..=> Str`) that has two main dimensions: `Degree` and `AdjForm`, where the last one is a combination of the parameters listed above. The reason to have the `Degree` parameter as a separate dimension is that a special type of adjectives `Adj` that just have positive forms is useful. It includes both non-degree adjective classes: possessive, like *мамин* (*mother's*), *лисий* (*fox'es*), and relative, like *русский* (*Russian*).

As a part of the language-independent API, the name `AdjDegree` denotes the adjective degree type for all languages, although each language has its own implementation. Maintaining parallelism among languages is rather straightforward at this stage, since the only thing shared is the name of

[1]The English *-er/more* and *-est/most* variations are exclusive, while in Russian both forms are valid.

a part of speech. A possible complication is that parsing with inflectionally rich languages can be less efficient compared to, for instance, English. This is because in GF all forms of a word are kept in the same declension table, which is convenient for generation, since GF is a generation-oriented grammar formalism. Therefore, the more forms there are, the bigger tables we have to store in memory, which can become an issue as the grammars grow and more languages are added (Dada and Ranta, 2006).

3 Inflection Paradigms and Lexicon

Besides word class declarations, morphology modules also contain functions defining common inflectional patterns (**paradigms**) and a lexicon. This information is language-specific, so fitting into the common API is not a consideration here. Paradigms are used to build the lexicon incrementally as new words are used in applications. A lexicon can also be extracted from other sources.

Unlike syntactic descriptions, morphological descriptions for many languages have been already developed in other projects. Thus, considerable efforts can be saved by reusing existing code. How easy we can perform the transformation depends on how similar the input and output formats are. For example, the Swedish morphology module is generated automatically from the code of another project, called Functional Morphology (Forsberg and Ranta, 2004). In this case the formats are very similar, so extracting is rather straightforward. However, this might not be the case if we build the lexicon from a very different representation or even from corpora, where post-modification by hand is simply inevitable.

A paradigm function usually takes one or more string arguments and forms a lexical entry. For example, the function `nGolova` describes the inflectional pattern for feminine inanimate nouns ending with *-a* in Russian. It takes the basic form of a word as a string (`Str`) and returns a noun (`CN` stands for Common Noun, see definition in section 4). Six cases times two numbers gives twelve forms, plus two inherent parameters `Animacy` and `Gender` (defined in section 2):

```
oper
 nGolova:  Str -> CN = \golova ->
   let golov = init golova in {
   s = table {
     SF Sg Nom => golov+"а";
     SF Sg Gen => golov+"ы";
     SF Sg Dat => golov+"е";
     SF Sg Acc => golov+"у";
     SF Sg Inst => golov+"ой";
     SF Sg Prepos => golov+"е";
     SF Pl Nom => golov+"ы";
     SF Pl Gen => golov;
     SF Pl Dat => golov+"ам";
     SF Pl Acc => golov+"ы";
     SF Pl Inst => golov+"ами";
     SF Pl Prepos => golov+"ах" };
   g = Fem;
   anim = Inanimate };
```

where `\golova` is a λ-abstraction, which means that the function argument of the type `Str` will be denoted as `golova` in the definition. The construction `let...in` is used to extract the word stem (`golov`), in this case, by cutting off the last letter (`init`). Of course, one could supply the stem directly, however, it is easier for the grammarian to just write the whole word without worrying what stem it has and let the function take care of the stem automatically. The table structure is simple – each line corresponds to one parameter value. The sign `=>` separates parameter values from corresponding inflection forms. Plus sign denotes string concatenation.

The **type signature** (`nGolova: Str -> CN`) and maybe a comment telling that the paradigm describes feminine inanimate nouns ending with *-a* are the only things the grammarian needs to know, in order to use the function `nGolova`. Implementation details (the inflection table) are hidden. The name `nGolova` is actually a transliteration of the Russian word *голова* (*head*) that represents nouns conforming to the pattern. Therefore, the grammarian can just compare a new word to the word *голова* in order to decide whether `nGolova` is appropriate. For example, we can define the word `mashina` (*машина*) corresponding to the English word *car*. Машина is a feminine, inanimate noun ending with *-a*. Therefore, a new lexical entry for the word *машина* can be defined by:

```
oper mashina = nGolova "машина" ;
```

Access via type signature becomes especially helpful with more complex parts of speech like verbs.

Lexicon and inflectional paradigms are language-specific, although, an attempt to build a general-purpose interlingua lexicon in GF has been made. Multilingual dictionary can work for words denoting unique objects like *the sun* etc., but otherwise, having a common lexicon interface does not sound like a very good idea or at least something one would like to start with. Normally, multilingual dictionaries have bilingual organization (Kellogg, 2005).

At the moment the resource grammar has an interlingua dictionary for, so called, closed word classes like pronouns, prepositions, conjunctions and numerals. But even there, a number of discrepancies occurs. For example, the impersonal pronoun *one* (`OnePron`) has no direct correspondence in Russian. Instead, to express the same meaning Russian uses the infinitive: *если очень захотеть, можно в космос улететь* (*if one really wants, one can fly into the space*). Note, that the modal verb *can* is transformed into the adverb *можно* (*it is possible*). The closest pronoun to *one* is the personal pronoun *ты* (*you*), which is omitted in the final sentence: *если очень захочешь, можешь в космос улететь*. The Russian implementation of `OnePron` uses the later construction, skipping the string (`s`), but preserving number (`n`), person (`p`) and animacy (`anim`) parameters, which are necessary for agreement:

```
oper OnePron: Pronoun = {
  s = "";
  n = Singular;
  p = P2;
  anim = Animate };
```

4 Syntax

Syntax modules describe rules for combining words into phrases and sentences. Designing a language-independent syntax API is the most difficult part: several revisions have been made as the resource coverage has grown. Russian is very different from other resource languages, therefore, it sometimes fits poorly into the common API.

Several factors have influenced the API structure so far: application domains, parsing algorithms and supported languages. In general, the resource syntax is built bottom-up, starting with rules for forming noun phrases and verb phrases, continuing with relative clauses, questions, imperatives, and coordination. Some textual and dialogue features might be added, such as contrasting, topicalization, and question-answer relations.

On the way from dictionary entries towards complete sentences, categories loose declension forms and, consequently, get more parameters that ”memorize” what forms are kept, which is necessary to arrange agreement later on. Closer to the end of the journey string fields are getting longer as types contain more complex phrases, while parameters are used for agreement and then left behind. Sentence types are the ultimate types that just contain one string and no parameters, since everything is decided and agreed on by that point.

Let us take a look at Russian nouns as an example. A noun lexicon entry type (`CN`) mentioned in section 3 is defined like the following:

```
param
 SubstForm = SF Number Case;
oper
 CN: Type = {
  s:  SubstForm => Str;
  g: Gender;
  anim: Animacy  };
```

As we have seen in section 3, the string table field `s` contains twelve forms. On the other hand, to use a noun in a sentence we need only one form and several parameters for agreement. Thus, the ultimate noun type to be used in a sentence as an object or a subject looks more like Noun Phrase (`NP`):

```
oper NP : Type = {
  s: Case => Str;
  Agreement: {
    n: Number;
    p: Person;
    g: Gender;
    anim: Animacy} };
```

which besides `Gender` and `Animacy` also contains `Number` and `Person` parameters (defined in section 2), while the table field `s` only contains six forms: one for each `Case` value.

The transition from `CN` to `NP` can be done via various intermediate types. A noun can get modifiers like adjectives – *красная комната* (*the red room)*, determiners – *много шума* (*much ado)*, genitive constructions – *герой нашего времени* (*a hero of our time)*, relative phrases – *человек, который смеётся* (*the man who laughs)*. Thus, the string field (`s`) can eventually contain more than one word. A noun can become a part of other phrases, e.g. a predicate in a verb phrase – *знание – сила* (*knowledge is power)* or a complement in a prepositional phrase – *за рекой, в тени деревьев* (*across the river and into the trees*).

The language-independent API has an hierarchy of intermediate types all the way from dictionary entries to sentences. All supported languages follow this structure, although in some cases this does not happen naturally. For example, the division between definite and indefinite noun phrases is not relevant for Russian, since Russian does not have any articles, while being an important issue about nouns in many European languages. The common API contains functions supporting such division, which are all conflated into one in the Russian implementation. This is a simple case, where Russian easily fits into the common API, although a corresponding phenomenon does not really exist.

Sometimes, a problem does not arise until the joining point, where agreement has to be made. For instance, in Russian, numeral modification uses different cases to form a noun phrase in nominative case: *три товарища* (*three comrades*), where the noun is in nominative, but *пять товарищей* (*five comrades*), where the noun is in genitive! Two solutions are possible. An extra non-linguistic parameter bearing the semantics of a numeral can be included in the `Numeral` type. Alternatively, an extra argument (`NumberVal`), denoting the actual number value, can be introduced into the numeral modification function (`IndefNumNP`) to tell apart numbers with the last digit between 2 and 4 from other natural numbers:

```
oper IndefNumNP: NumberVal ->
         Numeral -> CN -> NP;
```

Unfortunately, this would require changing the language-independent API (adding the `NumberVal` argument) and consequent adjustments in all other languages that do not need this information. Note, that `IndefNumNP`, `Numeral`, `CN` (Common Noun) and `NP` (Noun Phrase) belong to the language-independent API, i.e. they have different implementations in different languages. We prefer the encapsulation version, since the other option will make the function more error-prone.

Nevertheless, one can argue for both solutions, which is rather typical while designing a common interface. One has to decide what should be kept language-specific and what belongs to the language-independent API. Often this decision is more or less a matter of taste. Since Russian is not the main language in the GF resource library, the tendency is to keep things language-specific at least until the common API becomes too restrictive for a representative number of languages.

The example above demonstrates a syntactic construction, which exist both in the language-independent API and in Russian although the common version is not as universal as expected. There are also cases, where Russian structures are not present in the common interface at all, since there is no direct analogy in other supported languages. For instance, a short adjective form is used in phrases like *мне нужна помощь* (*I need help*) and *ей интересно искусство* (*she is interested in art*). In Russian, the expressions do not have any verb, so they sound like *to me needed help* and *to her interesting art*, respectively. Here is the function `predShortAdj` describing such adjective predication[2] specific to Russian:

```
oper predShortAdj: NP -> Adj ->
 NP -> S = \I, Needed, Help -> {
   s = let {
     toMe  = I.s ! Dat;
     needed  = Needed.s !
      AF Short Help.g Help.n;
     help = Help.s ! Nom
    } in
    toMe ++ needed ++ help };
```

`predShortAdj` takes three arguments: a non-degree adjective (`Adj`) and two noun phrases (`NP`) that work as a predicate, a subject and an object in the returned sentence (`S`). The third line indicates that the arguments will be denoted as `Needed`, `I` and `Help`, respectively (λ-abstraction). The sentence type (`S`) only contains one string field `s`. The construction `let...in` is used to first form the individual words (`toMe`, `needed` and `help`) to put them later into a sentence. Each word is produced by taking appropriate forms from inflection tables of corresponding arguments (`Needed.s`, `Help.s` and `I.s`). In the noun arguments `I` and `Help` dative and nominative cases, respectively, are taken (`!`-sign denotes the selection operation). The adjective `Needed` agrees with the noun `Help`, so `Help`'s gender (`g`) and number (`n`) are used to build an appropriate adjective form (`AF Short Help.g Help.n`). This is exactly where we finally use the parameters from `Help` argument of the type `NP` defined above. We only use the declension tables from the argu-

[2]In this example we disregard adjective past/future tense markers *было/будет*.

ments `I` and `Needed` – other parameters are just thrown away. Note, that `predShortAdj` uses the type `Adj` for non-degree adjectives instead of `AdjDegree` presented in section 2. We also use the `Short` adjective form as an extra `Case`-value.

5 An Example Application Grammar

The purpose of the example is to show similarities between the same grammar written for different languages using the resource library. Such similarities increase the reuse of previously written code across languages: once written for one language a grammar can be ported to another language relatively easy and fast. The more language-independent API functions (names conventionally starting with a capital letter) a grammar contains, the more efficient the porting becomes.

We will consider a fragment of `Health` – a small phrase-book grammar written using the resource grammar library in English, French, Italian, Swedish and Russian. It can form phrases like *she has a cold and she needs a painkiller*. The following categories (`cat`) and functions (`fun`) constitute language-independent abstract syntax (domain semantics):

```
cat
 Patient; Condition;
 Medicine; Prop;
fun
 ShePatient:    Patient;
 CatchCold:     Condition;
 PainKiller:    Medicine;
 BeInCondition: Patient ->
   Condition -> Prop;
 NeedMedicine:  Patient ->
   Medicine  -> Prop;
 And:   Prop -> Prop -> Prop;
```

Abstract syntax determines the class of statements we are able to build with the grammar. The category `Prop` denotes complete propositions like *she has a cold*. We also have separate categories of smaller units like `Patient`, `Medicine` and `Condition`. To produce a proposition one can, for instance, use the function `BeInCondition`, which takes two arguments of the types `Patient` and `Condition` and returns the result of the type `Prop`. For example, we can form the phrase *she has a cold* by combining three functions above:

```
BeInCondition
  ShePatient CatchCold
```

where `ShePatient` and `CatchCold` are constants used as arguments to the function `BeInCondition`.

Concrete syntax translates abstract syntax into natural language strings. Thus, concrete syntax is language-specific. However, having the language-independent resource API helps to make even a part of concrete syntax shared among the languages:

```
lincat
 Patient       = NP;
 Condition     = VP;
 Medicine      = CN;
 Prop          = S;
lin
 And           = ConjS;
 ShePatient    = SheNP;
 BeInCondition = PredVP;
```

The first group (`lincat`) tells that the semantic categories `Patient`, `Condition`, `Medicine` and `Prop` are expressed by the resource linguistic categories: noun phrase (`NP`), verb phrase (`VP`), common noun (`CN`) and sentence (`S`), respectively. The second group (`lin`) tells that the function `And` is the same as the resource coordination function `ConjS`, the function `ShePatient` is expressed by the resource pronoun `SheNP` and the function `BeInCondition` is expressed by the resource function `PredVP` (the classic `NP_VP->S` rule). Exactly the same rules work for all five languages, which makes the porting trivial[3]. However, this is not always the case.

Writing even a small grammar in an inflectionally rich language like Russian requires a lot of work on morphology. This is the part where using the resource grammar library may help, since resource functions for adding new lexical entries are relatively easy to use. For instance, the word *painkiller* is defined similarly in five languages by taking a corresponding basic word form as an argument to an inflection paradigm function:

```
-- English:
PainKiller = regN "painkiller";

-- French:
PainKiller = regN "calmant";

-- Italian:
PainKiller = regN "calmante";
```

[3]Different languages can actually share the same code using GF parameterized modules (Ranta, to appear)

```
-- Swedish:
PainKiller = regGenN
  "smärtstillande" Neut;

-- Russian:
PainKiller = nEe "обезболивающее";
```

The `Gender` parameter (`Neut`) is provided for Swedish.

In the remaining functions we see bigger differences: the idiomatic expressions *I have a cold* in French, Swedish and Russian is formed by adjective predication, while a transitive verb construction is used in English and Italian. Therefore, different functions (`PosA` and `PosTV`) are applied. `tvHave` and `tvAvere` denote transitive verb *to have* in English and Italian, respectively. `IndefOneNP` is used for forming an indefinite noun phrase from a noun in English and Italian:

```
-- English:
CatchCold = PosTV tvHave
 (IndefOneNP (regN "cold"));

-- Italian:
CatchCold = PosTV tvAvere
 (IndefOneNP (regN "raffreddore"));

-- French:
CatchCold = PosA (regA "enrhumé")

-- Swedish:
CatchCold = PosA
(mk2A "förkyld" "förkylt");

-- Russian:
CatchCold = PosA
 (adj_yj "простужен");
```

In the next example the Russian version is rather different from the other languages. The phrase *I need a painkiller* is a transitive verb predication together with complementation rule in English and Swedish. In French and Italian we need to use the idiomatic expressions *avoir besoin* and *aver bisogno*. Therefore, a classic `NP_VP` rule (`PredVP`) is used. In Russian the same meaning is expressed by using adjective predication defined in section 4:

```
--English:
NeedMedicine pat med = predV2
 (dirV2 (regV "need"))
  pat (IndefOneNP med);

-- Swedish:
NeedMedicine pat med = predV2
  (dirV2 (regV "behöver"))
  pat (DetNP nullDet med);

-- French:
NeedMedicine pat med = PredVP
 pat (avoirBesoin med);

-- Italian:
NeedMedicine pat med = PredVP
 pat (averBisogno med);

-- Russian:
NeedMedicine pat med =
 predShortAdj pat
 (adj_yj "нужен") med;
```

Note, that the medicine argument (`med`) is used with indefinite article in the English version (`IndefOneNP`), but without articles in Swedish, French and Italian. As we have mentioned in section 4, Russian does not have any articles, although the corresponding operations exist for the sake of consistency with the language-independent API.

`Health` grammar shows that the more similar languages are, the easier porting will be. However, as with traditional translation the grammarian needs to know the target language, since it is not clear whether a particular construction is correct in both languages, especially, when the languages seem to be very similar in general.

6 Conclusion

GF resource grammars are general-purpose grammars used as a basis for building domain-specific application grammars. Among pluses of using such grammar library are guaranteed grammaticality, code reuse (both within and across languages) and higher abstraction level for writing application grammars. According to the ”division of labor” principle, resource grammars comprise the necessary linguistic knowledge allowing application grammarians to concentrate on domain semantics.

Following Chomsky's universal grammar hypothesis (Chomsky, 1981), GF multilingual resource grammars maintain a common API for all supported languages. This is implemented using

GF's mechanism of separating between abstract and concrete syntax. Abstract syntax declares universal principles, while language-specific parameters are set in concrete syntax. We are not trying to answer the general question what constitutes universal grammar and what beyond universal grammar differentiates languages from one another. We look at GF parallel resource grammars as a way to simplify multilingual applications.

The implementation of the Russian resource grammar proves that GF grammar formalism allows us to use the language-independent API for describing sometimes rather peculiar grammatical variations in different languages. However, maintaining parallelism across languages has its limits. From the beginning we were trying to put as much as possible into a common interface, shared among all the supported languages. Word classes seem to be rather universal at least for the eleven supported languages. Syntactic types and some combination rules are more problematic. For example, some Russian rules only make sense as a part of language-specific modules while some rules that were considered universal at first are not directly applicable to Russian.

Having a universal resource API and grammars for other languages has made developing Russian grammar much easier comparing to doing it from scratch. The abstract syntax part was simply reused. Some concrete syntax implementations like adverb description, coordination and subordination required only minor changes. Even for more language-specific rules it helps a lot to have a template implementation that demonstrates what kind of phenomena should be taken into account.

The GF resource grammar development is mostly driven by application domains like software specifications (Burke and Johannisson, 2005), math problems (Caprotti, 2006) or transport network dialog systems (Bringert et al., 2005). The structure of the resource grammar library is continually influenced by new domains and languages. The possible direction of GF parallel resource grammars' development is extending the universal interface by domain-specific and language-specific parts. Such adaptation seems to be necessary as the coverage of GF resource grammars grows.

Acknowledgements

Thanks to Professor Arto Mustajoki for fruitful discussions and to Professor Robin Cooper for reading and editing the final version of the paper. Special thanks to Professor Aarne Ranta, my supervisor and the creator of GF.

References

B. Bringert, R. Cooper, P. Ljunglöf, and A. Ranta. 2005. Multimodal Dialogue System Grammars. In *DIALOR'05, Nancy, France*.

D.A. Burke and K. Johannisson. 2005. Translating Formal Software Specifications to Natural Language / A Grammar-Based Approach. In *LACL 2005, LNAI* 3402, pages 51–66. Springer.

M. Butt, T. H. King, M.-E. Ni no, and F. Segond, editors. 1999. *A Grammar Writer's Cookbook*. Stanford: CSLI Publications.

O. Caprotti. 2006. WebALT! Deliver Mathematics Everywhere. In *SITE 2006, Orlando, USA*.

N. Chomsky. 1981. *Lectures on Government and Binding: The Pisa Lectures*. Dordrecht, Holland: Foris Publications.

A. E. Dada and A. Ranta. 2006. Implementing an arabic resource grammar in grammatical framework. At 20th Arabic Linguistics Symposium, Kalamazoo, Michigan. URL: `www.mdstud.chalmers.se/~eldada/paper.pdf`.

M. Forsberg and A. Ranta. 2004. Functional morphology. In *ICFP'04*, pages 213–223. ACM Press.

M. Kellogg. 2005. Online french, italian and spanish dictionary. URL: `www.wordreference.com`.

C. Pollard and I. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press.

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

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

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

M.A. Shelyakin. 2000. *Spravochnik po russkoj grammatike (in Russian)*. Russky Yazyk, Moscow.

S. Starostin. 2005. Russian morpho-engine on-line. URL: `starling.rinet.ru/morph.htm`.

T. Wade. 2000. *A Comprehensive Russian Grammar*. Blackwell Publishing.