Thesis for the Degree of Doctor of Philosophy

Language Engineering in Grammatical Framework (GF)

Janna Khegai

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, September 2006

Language engineering in Grammatical Framework (GF)
Janna Khegai
ISBN 91-7291-813-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie Nr 2494
ISSN 0346-718x

Technical Report no.19D
Department of Computer Science and Engineering
Language Technology Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone + 46 (0)31-772 1000

Printed at Chalmers, Göteborg, Sweden, 2006

Language engineering in Grammatical Framework (GF)

JANNA KHEGAI
Department of Computer Science
Chalmers University of Technology and Göteborg University

Abstract

The basis for the work presented is Grammatical Framework (GF) — a grammar formalism based on type theory. It is also a powerful language processor that provides a convenient framework for various multilingual applications. The primary concern of this thesis is the usage of GF as a piece of software. The main results are:

- Implementing a syntax editor, which provides a graphical user interface (GUI) for the command-line GF core.
- Writing the Russian resource grammar that takes care of the most basic morphological and syntactic rules and serves as a standard library for building application grammars (describing sublanguage domains) in Russian.

These results contribute to language engineering in GF on two different levels:

- Author level (end-user) — constructing documents in natural languages.
- Grammarian level — building a grammar description, which is later used on the author level. One can also distinguish between application and resource grammars. An application grammar focuses of a particular sub-language domain, while resource grammar is a general-purpose grammar that forms a basis for application grammars.

Keywords: Russian resource grammar, syntax editing, multilingual authoring, graphical user interface (GUI), interlingua, natural language processing (NLP), computational linguistics, machine translation (MT)

Acknowledgements[1]

I would like to thank my supervisor, Aarne Ranta for giving me a project in the first place, for his careful guidance in the beginning and for the encouragement for independence in the end. I feel very lucky to work with GF – a full-cycle machine translation engine: on one hand, it lets you focus on a specific area; on the other hand, it gives you the satisfaction of seeing the final result where all the pieces come together.

Thanks a lot to Arto Mustajoki and his colleagues from the Slavonic and Baltic Department at the University of Helsinki for fruitful discussions about Russian. Also thanks to Lars Borin for his insightful comments on the Russian resource grammar. Many thanks to my opponent Lauri Carlson and the members of the grading committee: Guy Perrier, Kuchi Prasad and Andrei Sabelfeld. Also thanks to Mats Wirén from Telia Research for sharing his knowledge about CLE and to the members of my advisory committee – Bengt Nordström and Devdatt Dubhashi who donated their time and ideas to the project as well as promoted publication of the results. Many thanks to all members of language technology group, especially: Robin Cooper, Peter Ljunglöf, Kristofer Johannisson, Markus Forsberg, Björn Bringert and Harald Hammarström for their help and professional advice.

I am grateful to my office mates: Angela Wallenburg, Wojciech Mostowski, Erik Kilborn, Kristofer Johannisson and Markus Forsberg for being excellent neighbours. I also want to express my appreciation to other PhD students whose support has been invaluable and made my time at Chalmers a lot of fun. My gratitude to all people at the department of Computing Science at Chalmers for the opportunity to study in a friendly and intellectually charged atmosphere.

Finally, I am happy to thank my friends and relatives outside the department, my husband Alexandre and particularly my daughter Vera for a break from my PhD studies as well as for some extra time to complete them.

[1]This thesis has been partially supported by: 1) "Interactive Language Technology" funded by Vinnova (2001-2004). Project nr P20018-2A. 2) "Grammars as Software Libraries" funded by Vetenskapsrådet (2006-2008).

List of Included Papers

This thesis is based on work contained in the following peer-reviewed conference papers:

Paper 1: Janna Khegai. "GF Parallel Resource Grammars and Russian". // In proceedings of Coling/ACL2006 (The joint conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics), pages 475-482, Sydney, Australia, July 2006.

Paper 2: Janna Khegai and Aarne Ranta. "Building and Using a Russian Resource Grammar in GF". // In Intelligent Text Processing and Computational Linguistics (CICLing-2004), Seoul, Korea. LNCS 2945, pages 38-41. Springer, 2004.

Paper 3: Janna Khegai. "Grammatical Framework (GF) for MT in sublanguage domains". // In proceedings of EAMT-2006 (11th Annual conference of the European Association for Machine Translation), pages 95-104, Oslo, Norway, June 2006.

Paper 4: Janna Khegai, Bengt Nordström and Aarne Ranta. "Multilingual Syntax Editing in GF". // In Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico. LNCS 2588, pages 453-464. Springer, 2003.

and technical reports:

Technical report A: Janna Khegai and Aarne Ranta. "GF Russian Resource Library". //Technical Report no. 2006-14, Department of Computing Science, Chalmers University of Technology, 2006.

Technical report B: Janna Khegai. "Syntax Editing in GF". // Technical Report no. 2006-15, Department of Computing Science, Chalmers University of Technology, 2006.

Contents

Introduction

As an interdisciplinary topic, Language Engineering has attracted the attention of both linguists (Computational Linguistics) and computer scientists (Natural Language Processing). The purpose of their joint efforts is building computationally rather than psychologically plausible representations of human language, i.e. a representation suitable for computer processing. This is done by formalizing the linguistic knowledge into grammar rules. A language for describing such rules is called a grammar formalism.

Grammatical Framework (GF) is a grammar formalism based on constructive Type Theory [25]. Together with its implementation it forms a framework for performing various Natural Language Processing (NLP) tasks. We will give a brief overview of the system in sections 1 and 2. For a systematic description of GF we refer to [30, 21, 32].

Section 3 contains an overview of the thesis. Related work is discussed in section 4. The chapter concludes with stating the main results (section 5) and future work (section 6).

1 The GF grammar formalism

GF originates from the tradition of logical frameworks, which give the possibility to define logical calculi that can be used for interactive theorem proving. This tradition is closely connected to the functional programming paradigm, whose programming style is very close to the language of mathematics. GF is implemented in the functional programming language Haskell. Logical frameworks implement Type Theory concepts and some of the frameworks even translate these concepts into user-friendly notations. Although the user can define new mathematical objects in a logical framework, the expressions that could be used for that purpose are limited to those hard-wired in the system. GF extends Type Theory into a grammar formalism by adding a notation for syntactic annotations.

The main features of the GF grammar formalism are:

- mapping between abstract and concrete syntax levels
- type system on both levels

The first property makes it natural to have multilingual grammars with a shared abstract part (interlingua) and different concrete parts for different languages. The type system on the abstract level allows us to verify the well-formedness of an input as well as resolve ambiguities using semantic information contained in the type of the input. The type system on the concrete level prevents run-time errors with grammars and forces grammaticality when using a resource grammar library.

A grammar in GF is defined in a declarative way using GF syntactic notation close to those used in functional programming languages. The definition consists of an abstract and a concrete part. Let us consider a simple grammar for writing letters in several languages. Here is a piece from the abstract part:

```
cat
  Letter ;
  Recipient ;
  Heading ;
  Message ;
  Ending ;

fun
  MkLetter : Heading -> Message -> Ending -> Letter ;
  NameHe, NameShe : String -> Recipient ;
  DearRec   : Recipient -> Heading ;
```

This introduces five types (categories) for different types of letter elements after the reserved word `cat` and four functions (rules) for constructing a letter after the reserved word `fun`. The first function says that a letter consists of a heading, a message and an ending. The second and third tell that a recipient can be formed from a string representing a name. The last function forms a *Dear* heading from a recipient argument. So far we describe the letter domain in a language-independent manner. Thus the abstract syntax is a sort of *interlingua* — a semantic (meaning) representation of the domain.

Language specific components are placed in the concrete part of the grammar. Here is a fragment from the English concrete syntax:

```
param
  Sex = masc | fem ;
  Num = sg | pl ;
  DepNum = depnum | cnum Num ;

lincat
  Letter     = {s : Str} ;
  Recipient  = {s : Str ; n : Num ; x : Sex} ;
  Heading    = {s : Str ; n : Num ; x : Sex} ;
```

```
  -- needs Author's & Recipient's Num and Sex:
  Message    = {s : DepNum => Sex => Num => Sex => Str} ;
  -- needs Recipient's Num and Sex:
  Ending     = {s : Num => Sex => Str ; n : DepNum ; x : Sex} ;

lin
  NameHe s   = {s = s.s  ; n = sg ; x = masc} ;
  NameShe s  = {s = s.s  ; n = sg ; x = fem} ;

  DearRec rec   = {s = "Dear" ++ rec.s ; n = rec.n ; x = rec.x} ;

  MkLetter head mess end = { s = head.s ++ "," ++ "&-" ++
      mess.s ! end.n ! end.x ! head.n ! head.x ++ "." ++ "&-" ++
      end.s ! head.n ! head.x };
```

Here we can see the correspondence between the abstract and the concrete part. Each category (and each function) introduced in the abstract part has a linearization in the concrete part (after the reserved word `lincat` and the reserved word `lin`, respectively). Category linearizations describe the type of the category declared in the abstract part for a concrete language. All categories have record types with one or more record fields.

For instance, the category `Letter` contains one field of the type String (`Str`). This basically means that a letter is a string. The categories `Recipient` and `Heading` have number (`n: Num`) and sex (`x: Sex`) fields besides the string field, where `Num` and `Sex` are parameters with values enumerated after the reserved word `param`. `n` and `x` are so called inherent parameters, which are fixed for every instance of `Recipient` and `Heading`.

The linearization type of the `Message` is a table of strings that depend on four parameters, namely, the letter author's and recipient's sexes (`Sex`) and numbers (`DepNum`, `Num`), because the message can have references to both the author and the recipient, for example: *I love you* or *you have been promoted to project managers*. The `DepNum` parameter is introduced for the cases where the author's number depends on the recipient's number, for instance in case of a spouse(s) letter. Thus, the content of the message does not determine these four parameters. As follows from the linearization types `Heading` and `Ending` they determine the sex and number parameters for the author and the recipient, respectively. According to the `Ending` type, the ending potentially needs to know the sex and number parameters of the recipient (spouse(s) letters), while `Heading` is independent.

These type definitions become more meaningful if we look at function linearizations. For example, the linearizations for the function `NameHe`, which returns the result of the type `Recipient`, says that the number of the resulting

`Recipient` is singular and the sex is masculine while for the function `NameShe` the sex will be feminine, which is also reflected in the names of the functions. The field `s` of `NameHe` and `NameShe` just contains the string, which they take as an argument. Thus, `NameShe "Mary"` gives as a result the following record of the type `Recipient`:

```
{s = "Mary"; n = sg; x = fem}
```

Similarly,

```
DearRec (NameShe "Mary)
```

will give the type `Heading` record

```
{s = "Dear Mary"; n = sg; x = fem}
```

since the `++`-sign in `DearRec` linearization means string concatenation and `rec.s` gets the value of the field `s` of the argument `rec`.

The linearizations for the function `MkLetter` puts together the string fields of the heading (`head`), the message (`mess`) and the ending (`end`). In case of the message and the ending we first choose an appropriate string from the corresponding tables. The exclamation sign (`!`) denotes the selection operation. For example, `end.s ! head.n ! head.x` means that we take a string from the ending table (`end.s`) that corresponds to the number and sex values of the heading number and sex fields (`head.n` and `head.x`, respectively). The "`&-`"-string stands for new line marker.

Now let us take a look at the Russian version of the grammar Letter, namely at the part, which differs from the English version:

```
DearRec rec = {s = regAdj "дорог" ! rec.n ! rec.x ++ rec.s ;
               n = rec.n ; x = rec.x} ;

regAdj :  Str -> Num => Sex => Tok =\s -> table {
 sg => table {masc => s + "ой"; fem => s + "ая" } ;
 pl => table {_ => s + "ие""}
 } ;
```

In the linearization of the function `DearRec` the values of the fields `n` and `x` are inherited from the argument `rec` similarly to the English version. The string field, however, is more complex due to inflecting of the word *dear* in Russian, whose inflection table is provided by the extra function `regAdj`. In order to choose the right form we use the number and the sex fields' values of the argument `rec`. Then we use the operator `!` to select the right values from the inflection table thus providing grammatically correct output.

As we can see, all language-specific things can be expressed in the concrete part, without disturbing the semantics of the abstract part. However, the abstract part should be refined enough to take into account all possible grammar variations in different languages. For instance, the English part in the fragment above does not really use the parameter `Sex`[2], since the word *dear* does not change. However, the two versions `NameShe` and `NameHe` of the same function are needed for compatibility with other languages, where the distinction between masculine and feminine forms is important.

A grammar definition is stored in a text file. Usually the abstract part and the concrete part for each language are stored in separate files. The `path` directive is used to access the content of other files that are not in the same directory as the original file. Grammar definitions are separated from the GF algorithmic core. They are loaded and compiled before usage. During the compilation a parser for the loaded grammar is generated. There is a number of user interfaces or modes in which GF can be run. The main functionality of the system comprises:

- constructing semantic trees for expressions covered by the loaded grammars using parsing
- constructing semantic trees for expressions covered by the loaded grammars using interactive syntax editing
- linearization of semantic trees for expressions covered by the loaded grammars

Translation is the combination of the first and the third operations above. Input language concrete syntax is used during the first step. Output language concrete syntax is used during the last step. The abstract syntax shared by all implemented languages is used over the whole translation procedure.

The abstract syntax represents a syntactic and partially semantic model of the natural language fragment we are trying to cover with a GF grammar. It defines categories and relationships among them very much similar to the manner of standard context-free rules, although using different notation suitable for expressing even context-dependent fenomena. Abstract syntax represents language independent properties of the described natural language domain.

Concrete syntax, represented by linearization rules introduces actual strings (words) from some natural language. The values returned by linearization can be not only strings, but also tables and records. This is especially important for expressing such language dependent features like morphological inflections without affecting the abstract part. Tables look very much like inflection tables in grammar books. Records remind of dictionary entries. For example, here is the description of the word *она* (*she*) in the Russian concrete syntax:

[2]The `Sex` is, however, needed in the English version in spouse(s) letters, for instance for an ending like *sincerely, your wife*.

```
oper pronOna:  Pronoun =
{ s = table {
PF Nom _ NonPoss => "она" ;
PF Gen No NonPoss => "её" ;
PF Gen Yes NonPoss => "неё" ;
PF Dat No NonPoss => "ей" ;
PF Dat Yes NonPoss => "ней" ;
PF Acc No NonPoss => "её" ;
PF Acc Yes NonPoss => "неё" ;
PF Inst No NonPoss => "ей" ;
PF Inst Yes NonPoss => "ней" ;
PF Prepos _ NonPoss => "ней" ;
PF _ _ (Poss _ ) => "её"
} ;
g = PGen Fem ;
n = Sg ;
p = P3 ;
} ;
```

Every morphological entry is represented as an operation definition starting with the reserved word `oper` followed by the name of the operation, here `pronOna`. The operation return type is specified after a semicolon, here `Pronoun`. Thus, the word *она (she)* in Russian is a pronoun. Pronoun type in Russian is a record with several fields that contain all the grammatical information belonging to the entry word: `s` (different word forms), `g`(gender), `n`(number) and `p`(person). The field `s` is a table that contains various inflection forms of the entry word just as a grammar book table does. The rest of the fields provide the information about: gender (`g`) — feminine (`PGen Fem`), number (`n`) — singular (`Sg`) and person (`p`) — third (`P3`) of the word *она (she)*. Similar descriptions of a word can be found in a dictionary.

Linearization and type checking are straightforward to implement following the techniques from logical frameworks and functional programming languages. Parsing is more difficult. The GF grammar formalism is stronger than context-free grammars. Referring to abstract syntax the context-free subclass of GF is equivalent to Parallel Multiple Context-Free Grammars (PMCFG), which has polynomial parsing complexity. Unrestricted GF grammars also contain higher-order functions and dependent types. A thorough investigation of GF's expressive power and parsing complexity together with optimized parsing algorithms can be found in [21].

2 GF in use

In the rest of the thesis we will talk about some usages of GF. GF provides a framework for producing high-quality automatic translation in limited domains, which determines the corresponding potential usage. However, since GF is an open-ended system that can be adapted to various applications we can distinguish between different kinds of usages.

Depending on their competence the GF users fall into one of the categories: author, grammarian, implementor.

On the author level the user is only allowed to write documents within a pre-existing grammar set. The graphical user interface (GUI) provided by Java GUI Syntax Editor described in paper 3, paper 4 and technical report B ensures that work on this level will not require any specific knowledge of GF.

The main purpose of the syntax editor is to construct a text simultaneously in several natural languages. The author does not have to know all the languages represented, but the GF system assures that if the output is correct in at least one of them, including the GF abstract language – language-independent semantic representation, then it will be syntactically and semantically correct in the rest of the languages. This reflects the idea of so-called multilingual authoring.

Unlike translation that processes complete natural language strings, syntax editor works with Abstract Syntax Trees (ASTs) and can handle incomplete objects by using temporary place-holders (meta-variables). Thus, ASTs not only come as the result of parsing, but also can be built directly using GF grammars. Corresponding strings in natural languages can be generated (linearized) from AST representation.

To build an AST, an interactive procedure of step-wise refinement of meta-variables (syntax editing) is used. At first one creates an object of a chosen type, for instance, a letter. At each step the author refines a tree node, for example, the heading, the message or the ending, by using a context-dependent menu that automatically suggests the valid options for each node. These options are generated from the underlying grammar. See paper 3, paper 4 and technical report B for syntax editing examples in GF.

On the grammarian level the user is able to add new grammars. It puts additional demands: the GF grammar formalism together with some linguistic knowledge. The grammarian creates a new grammar file in a text editor using the GF grammar formalism. However, the grammar development cost can be lowered by using the resource grammar library, which takes care of the standard grammatical rules. Paper 1, paper 2 and technical report A are devoted to the development of such a resource grammar library for Russian.

On these two levels we have some control for example over parsing algorithms to be used. However, the core of the system with parsing, linearization and other algorithms are hidden. Parsers for user grammars are generated automatically. Finally, an implementor is supposed to be a programmer with fair knowledge

of computer science in general and the GF system in particular. GF is a really inter-disciplinary project.

We are aiming to create an Integrated Development Environment (IDE) for the GF language that contains tools for users on different levels. GF is constantly developing, so the system status described is subject to change. For example, one of the recent additions is the GF embedded interpreter [4] – a library written purely in Java that allows for using GF parsing and linearization functionality without running the main GF system in the background. This thesis is a collection of more or less independent reports describing several projects within the GF system reflecting the progress towards our goal.

3 Thesis overview

The thesis consists of four refereed conference articles complemented, in order to provide more technical details, with two technical reports.

Paper 1: GF Parallel Resource Grammars and Russian

The paper shares the author's experience in implementing a Russian resource grammar in GF. It describes the main Russian modules trying to answer the question how well Russian fits into the common language-independent interface shared with the other supported languages.

The paper is published in proceedings of Coling/ACL-2006 (The joint conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics), pages 475-482, Sydney, Australia, July 2006.

Paper 2: Building and Using a Russian Resource Grammar in GF

The paper discusses the general structure of the GF resource grammar library and its usage for writing application grammars using Russian as an example.

The paper was written together with Aarne Ranta and was published in Intelligent Text Processing and Computational Linguistics (CICLing-2004), Seoul, Korea. LNCS 2945, pages 38-41. Springer, 2004.

The author's own contribution is implementing a Russian resource grammar and writing most of the paper.

Paper 3: Grammatical Framework (GF) for MT in Sublanguage Domains

The paper presents the GF system as a platform for building MT applications for natural sublanguage domains. It contains a demo of the recent version of GF Syntax Editor.

The paper is published in proceedings of EAMT-2006 (11th Annual conference of the European Association for Machine Translation), pages 95-104, Oslo, Norway, 2006.

Paper 4: Multilingual Syntax Editing in GF

A paper that describes the GF application to multilingual authoring and demonstrates GF syntax editor's functionality.

The paper was written together with Aarne Ranta and Bengt Nordström and was published in Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico. LNCS 2588, pages 453-464. Springer, 2003.

The author's own contribution is implementing the GUI (Graphical User Interface) for GF syntax editor and writing a part of the paper.

Technical reports

Two technical reports provide implementation details for the curious reader:

Technical report A complements the papers 1 and 2. It describes the implementation details of the Russian resource grammar. Some parts of the report have been automatically translated from the GF grammar format into the latex format using the *gfdoc* tool. The report is written together with Aarne Ranta. The author's own contribution is writing the Russian-specific part.

Technical report B complements the papers 3 and 4. It tries to address the theoretical issues behind the implementation. It also contains a user-manual for GF Java user interface and briefly tells about implementing Gramlets – a specialized version of the GF syntax editor running on PDA and as an applet.

4 Related work

Since GF itself has many different aspects and the present work also deals with several GF issues we divide the related work accordingly. The resource part and the authoring part, combined in GF, do not accompany each other in the related projects. Therefore, it is natural to consider these projects separately.

4.1 Multilingual authoring

The idea of developing the user interface in Java programming language comes from the CtCoq system [39].

GF applies formal language techniques to natural languages. Syntax editing procedure is borrowed from proof editors used for interactive theorem proving and pretty-printing of the proofs (Alf [24], Lego [22, 1]), which, in turn, originate from earlier systems like Mentor [9] and Cornell program synthesizer [36]. Constructing a proof in a proof editor corresponds to constructing an abstract syntax tree in GF. The concrete part is, however, missing from proof editors, since the proofs are usually expressed in a symbolic language of mathematics.

The multilingual authoring approach is similar to the one in WYSIWYM tool [37, 27]. There, Multilingual Natural Language Generation from a semantic knowledge base expressed in a formal language (non-linguistic source) is opposed to Machine Translation (MT) (linguistic source).

The language-independent knowledge engineering (building a knowledge diagram) in WYSIWYM corresponds to the construction of an abstract syntax tree in GF. The refinement entities called anchors in WYSIWYM correspond to the GF meta-variables. Refinement steps are performed by choosing from the list of available options in both cases. Text generated from the current object in several languages (English, French and Italian for WYSIWYM) are shown to the user while editing. The language-independent ontology (domain model, terminology) in WYSIWYM corresponds to a grammar (Abstract part) in GF.

Even the architecture of one of the WYSISYM implementations DRAFTER-II [28] reminds that of GF in a way that the GUI part is separated from the processing engine: Prolog is used for both ontology description and generation while the GUI is written in CLIM (Common Lisp Interface Manager).

However, unlike WYSIWYM the GF architecture has one more separation, namely a special language (GF grammar formalism) built upon the main implementation language Haskell. This makes GF more generic compared to WYSIWYM, where the ontology of concepts is hard-wired. The WYSIWYM user is not supposed to change the ontology coverage, he is only allowed to work on the author level. By contrast, in GF, writing one's grammars is one of the main features and even supporting tools like the resource grammar library are provided for the grammarian.

Besides the design issues the underlying non-linguistic models in WYSIWYM and GF are quite different in nature. The GF model is more semantic oriented, while WYSIWYM is focused on syntactic information. Syntax is more developed and even stylistic issues are addressed in the recent WYSIWYM applications [29]. However, a clear semantic representation is work in progress. The model currently used is mostly of syntactic nature. It consists of dozens of parameters (features) for each sentence containing both syntactic (tense, modality etc.) and semantical (actual verb) information.

GF is more versatile in one more respect compared to WYSIWYM. For every grammar not only the generator is produced, but also a parser. Therefore, the author is allowed to type his input provided that it conforms to the grammar. This is considered useful for multilingual authoring applications because typing can speed up the tedious syntax editing procedure. Thus, GF syntax editor is an example of, so called pluralistic editor [38] that supports both text and tree views.

GF was one of the sources of inspiration for an XML-based multilingual document authoring application for pharmaceutical domain developed at XRCE [5, 10]. The grammar formalism used in this system is called Interaction Grammars (IG)[3]. Like the GF grammar language, IG has a separation between the language-independent interlingua (abstract syntax in GF)and parallel realization grammars (concrete syntax in GF) for the languages represented (English and French). As GF, 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. It is based on Definite Clause Grammars — a unification-based extension of context-free grammars, which has a built-in implementation in Prolog.

4.2 Resource grammars

The resource grammar library is related to the Core Language Engine (CLE) project later used in Spoken Language Translator (SLT) system for Air Travel Information System (ATIS) domain [33].

Like the resource grammars in GF the CLE grammars aimed to be domain-independent, however they were trained upon and built with the ATIS corpus in mind. Domain vocabulary contains around 1000-2500 words. 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.

The SLT processing modules, generic in nature, are based on large, linguistically motivated grammars. However, the SLT system uses both grammatical (hand-coded rules) and statistical knowledge (trained preferences).

Quasi (scope-neutral) Logical Form (QLF) – a feature-based formalism is used for representing language structures. Due to quality-robustness trade-off the QLF formalism is deeper than surface constituent trees (for better quality), but captures only grammatical relations. It represents linguistic meaning, but not yet an interlingua, since robust parsing would be problematic in this case.

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.

[3]Not to be confused with Interaction Grammars in [26].

- bilingual transfer rules.

Both sets are specified in [33] 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 GF is similar to the transfer rules set (more than one thousand rules for each language pair), since GF is an interlingua system.

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). CLE contains some domain-specific rules like noun phrases for various flight codes etc. Time and date expressions are also treated specially in the CLE. The same phenomena are not treated in the same way. For example, prepositional phrases are handled by complements of verbs types (`V2` and `V3`) and as adverbs (`Adv`) in GF, so that phrases like *in the house* and *here* belong to the same category.

Verb phrase discontinuous constituents are handled by combining the record fields, while there is a special set of "movement" rules responsible for word order in the CLE. For example, to process the utterance *Are we men?* we need to use the following four rules:

```
S:[inv=y] -> V:[subcat=List]                 -- "are"
             NP                              -- "we"
             VP: [svi=movedv:[subcat=List]] -- "men"

VP -> V:[subcat=COMPS]
      COMPS

V:[subcat=[comp:COMP],svi=movedv:[subcat=[comp:COMP]]] -> []

COMP:[subjform=normal] -> NP
```

where the first rule is a special rule that takes care of the inverted word order. It says that the verb (`V`)[*are*], which is supposed to be a part of the verb phrase(`VP`) [*are men*] will go first. The rule also duplicates the information about this fronted verb in the `VP`'s feature `svi`, so it can be later used by the second and the third rules. The second rule forms a verb phrase taking a verb (`V`) and its complement (`COMP`), which are in turn are formed by the third and the forth rules. The third rule indicates that due to inverted word order (the feature `svi`) the verb phrase will appear in the sentence without the verb (since it is already placed in the front of the sentence). Notice that in case of the non-inverted word order we would need to use different rules for `V` and `S`. The fourth rule gives a complement to the verb phrase expressed by a noun phrase (`NPs`) [*men*].

In case of GF the word order issue is taken care of only on the sentence level by applying a generic rule for making clauses:

```
mkClause : Str -> Agr -> VP -> Clause =
  \subj,agr,vp -> {
    s = \\t,a,b,o =>
      let
        verb  = vp.s ! t ! a ! b ! o ! agr ;
        compl = vp.s2 ! agr
      in
      case o of {
        ODir   => subj ++ verb.fin ++ vp.ad ++ verb.inf ++ compl ;
        OQuest => verb.fin ++ subj ++ vp.ad ++ verb.inf ++ compl
        }
  } ;
```

For example, in the rule above in the lines four and five we can see that the verb phrase (`vp`) [*be a man*] is used twice: in the verb `verb` [*are*] and in the complement `compl` [*men*], i.e. the verb phrase is discontinuous: different parts of the verb phrase are used in different parts of the sentence. The structure of the verb phrase is the following:

```
VP : Type = {
  s   : Tense => Anteriority => Polarity => Order => Agr =>
        {fin, inf : Str} ;
  ad  : Str ;
  s2  : Agr => Str
  } ;
```

By having a structure inside a verb phrase we avoid 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. For example, in the sentence *Are we men?* discussed above the word *men* should be in plural form, since it needs to agree with the subject [*we*] (in the rule denoted by `subj`). In GF this is done by selecting (using the exclamation mark `!`) the plural form from the table `vp.s`, where `agr` comprises the number of the subject, i.e. in this case – plural. In CLE we need to apply rules to the basic word form in order to get various form of the word, see the example below.

Another difference is that the whole inflection pattern of a word (according to several parameters) is put in one table in GF (see section 1), while several independent rules are used to express a similar pattern in CLE. In CLE one

rule can only take care of one parameter at a time. For example, here are the fragments of two morphological rules that take care of standard French adjective inflection:

```
adj:[..., num=p, gen=G] -->
    adj:[..., num=s, gen=G],
    [s].

adj:[..., num=s, gen=f] -->
    adj:[..., num=s, gen=m],
    [e].
```

The first one is responsible for the number parameter. It says that the plural form is formed by adding *-s* to the singular form. The second rule is responsible for inflecting according to gender. It says that the feminine form is formed by adding *-e* to the masculine form. As we can see, number and gender variations are described separately. So in order to get a plural feminine form we need to apply the second and the first rules consequently. Choosing from the inflection table in GF does not have such restrictions, i.e. all the inflection forms of a word are described in one rule (function).

GF application grammars can be considered as specializations of general-purpose resource grammars. However, unlike CLE, the specialization is done by hand-writing an application grammar, while a CLE specialized grammar is "trained" with a corpus. Application grammar writing in GF becomes more automatic by using the example-based grammar writing technique. The idea is to use parsing with resource grammars to automatically define linearizations for application grammar rules. For example, to linearize a rule for expressing phrases like *X chases Y* in the animal application grammar:

```
fun Answer : Entity -> Action -> Entity -> Phrase ;
```

one can just provide an example of similar grammatical construction like *the woman loves men*:

```
lin Answer woman_N love_V2 man_N = in Phr "the woman loves men" ;
```

and the linearization will be automatically derived. Note that the words used in the example (*woman*, *to love*, *man*) are from the resource lexicon and have the types corresponding to the linearization types (`N`, `V2`) of function arguments, i.e. in case of `Answer`-function the categories from the application grammar are linearized using the following resource grammar categories:

```
lincat
  Phrase = Phr ;
  Entity = N ;
  Action = V2 ;
```

Although GF is essentially an interlingua system, writing application grammar rules using the resource grammars can be seen as compile-time transfer, since different resource structures are mapped to express the same meaning in different languages. Some run-time transfer has also been introduced recently, see [3, 2].

The differences between CLE and GF are partly due to design decisions, partly hereditary to formalism's expressive means. Despite these differences 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 Results

Our general goal is to improve the GF (re-)usability and portability as an application as well as a piece of software.

We use the term portability in two senses:

Multi-lingual

The GF grammar language was used for building grammars in Russian, which offers a typologically interesting case study.

The main result in this area is the Russian resource grammar library adapted from the similar libraries for other languages. Such libraries allow the grammarian to write multilingual application grammars faster by reuse of the previously written, widely applicable code. Another advantage is that the resource grammars are tested and, therefore, guarantee that all the library functions are grammatically correct. Grammatical correctness is inherited to application grammars, which is guaranteed by type-checking. GF multilingual resource grammars have been used in projects like WebALT [6] (partly commercial) and KeY [13].

Some example application grammars in Russian were also written both from scratch and using the resource library.

Using Russian in GF also required fixing some technical issues regarding the Cyrillic alphabet.

Multi-platform

The platform portability part includes writing the Java GUI Syntax Editor, which provides a graphical user interface for the GF main system written in Haskell, as well as working on so called Gramlets – pure Java programs suitable for PDA and also able to work as applets on WWW.

Using Java programming language makes the code written able to be run on several platforms (including UNIX, Windows and Mac) without recompilation.

GF (including the Java GUI Syntax Editor) is extended[4] (by Hans-Joachim Daniels) and integrated (by Kristofer Johannisson) as a plug-in in the KeY project – a tool for formal software specification and verification based on the Together commercial CASE (Computer-Aided Software Engineering) tool [13], which is also implemented in Java. GF is called from the KeY together with a UML diagram. The specification authored in the Java GUI Syntax Editor is sent back. This allows us to look at GF in general and the Java GUI Syntax Editor in particular as a module or a supplement to another system, thus making GF a reusable piece of software.

Java GUI together with the communication protocol (XML) turn out to be quite adaptable for applications other than syntax editing. In Fig. 1 we can see the Numerals application showing numbers in dozens of languages, which was adapted from the Java GUI Syntax Editor in a couple of hours without any changes on the GF side.

6 Future work

So far the GF grammars has been developed interactively according to the build-try-improve circular model. Grammar writing is done in a common text-editor, while compiling and testing using the command-line GF mode or the Java GUI. GF Integrated Development Environment (IDE) specially designed for writing GF grammars is work in progress [19]. Unlike the Java GUI syntax editor a grammar editing tool belongs to a higher — grammarian — user level.

The purpose of GF IDE is pragmatic: it uses its knowledge of GF to help the grammarian in preparing correct GF code (application grammars). It can also save efforts in learning the non-trivial grammar formalism, since otherwise substantial training is needed even for simple grammar writing. Some of the GF IDE features are:

- Systematic treatment of "exotic" languages. UTF-8 encoding is used for languages with non-latin alphabets. The system recognizes and properly displays non-latin characters automatically.

- Example-based (example-based mode), menudriven (tree editing mode) grammar development. It saves time for scrolling the resource library files by hand and helps avoiding small syntactic mistakes and type-errors that can be automatically detected.

- Lexicon extension on-the-fly. Parsing with resource grammar may result in parser failure. 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.

[4] Among other things HTML support has been added [8]

Figure 1: Numerals GUI displays numbers in dozens of languages. Adapted from the Java Syntax Editor GUI without disturbing the GF core system.

The main function of GF IDE is interactive grammar localization for the languages supported by the resource grammar library.

Another project is enriching Gramlets syntax editing functionality with parsing functionality provided by GF Embedded Interpreter [4] – a pure Java library that implements a restricted version of the GF language processor.

Paper 1.
GF Parallel Resource Grammars and Russian

Janna Khegai. "GF Parallel Resource Grammars and Russian". In proceedings of Coling/ACL-2006 (The joint conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics), pages 475-482, Sydney, Australia, July 2006.

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+"a";
    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.md stud.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.

Paper 2.
Building and Using a Russian Resource Grammar in GF

Janna Khegai and Aarne Ranta. "Building and Using a Russian Resource Grammar in GF". In Intelligent Text Processing and Computational Linguistics (CICLing-2004), Seoul, Korea. LNCS 2945, pages 38-41. Springer, 2004.

Building and Using a Russian Resource Grammar in GF

Janna Khegai and Aarne Ranta

Department of Computing Science
Chalmers University of Technology and Gothenburg University
SE-41296, Gothenburg, Sweden
{janna, aarne}@cs.chalmers.se

Abstract. Grammatical Framework (GF) [5] is a grammar formalism for describing formal and natural languages. An application grammar in GF is usually written for a restricted language domain, e.g. to map a formal language to a natural language. A resource grammar, on the other hand, aims at a complete description of a natural languages. The language-independent grammar API (Application Programmer's Interface) allows the user of a resource grammar to build application grammars in the same way as a programmer writes programs using a standard library. In an ongoing project, we have developed an API suitable for technical language, and implemented it for English, Finnish, French, German, Italian, Russian, and Swedish. This paper gives an outline of the project using Russian as an example.

1 The GF Resource Grammar Library

The Grammatical Framework (GF) is a grammar formalism based on type theory [5]. GF grammars can be considered as programs written in the GF grammar language, which can be compiled by the GF program. Just as with ordinary programming languages, the efficiency of programming labor can be increased by reusing previously written code. For that purpose standard libraries are usually used. To use the library a programmer only needs to know the type signatures of the library functions. Implementation details are hidden from the user.

The GF resource grammar library [4] is aimed to serve as a standard library for the GF grammar language. It aims at fairly complete descriptions of different natural languages, starting from the perspective of linguistics structure rather the logical structure of applications. The current coverage is comparable with, but still smaller than, the Core Language Engine (CLE) project [2].

Since GF is a multilingual system the library structure has an additional dimension for different languages. Each language has its own layer, produced by visible to the linguist grammarian. What is visible to the application grammarian is a an API (Application Programmer's Interface), which abstracts away from linguistic details and is therefore, to a large extent, language-independent. The module structure of a resource grammar layer corresponding to one language is shown in Fig. 1. Arrows indicate the dependencies among the modules.

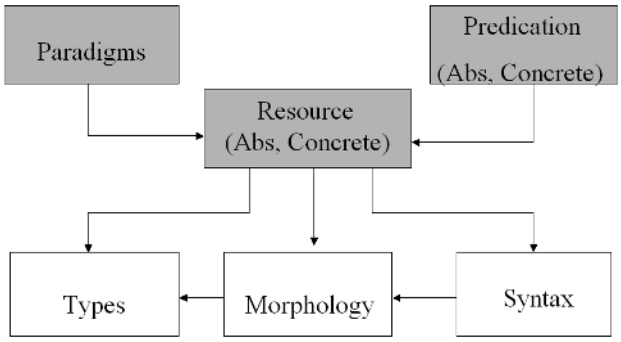

Fig. 1. The resource grammar structure (main modules). One language layer. Shadowed boxes represent high-level of interface modules. White boxes represent low-level or implementation modules. Arrows show the dependencies.

The Russian grammar was written after grammars for English, Swedish, French and German. The language-independent modules, defining the coverage of the resource library, were therefore ready. The task was to instantiate these modules for Russian. As a reference for Russian language, [3, 6, 7] were used.

2 An Example: Arithmetic Grammar

Here we consider some fragments from a simple arithmetic grammar written using the Russian resource grammar library, which allows us to construct statements like *one is even* or *the product of zero and one equals zero*.

The abstract part describes the meaning captured in this arithmetic grammar. This is done by defining some categories and functions:

```
cat
  Prop ;                      -- proposition
  Dom ;                       -- domain of quantification
  Elem Dom ;                  -- individual element of a domain
fun
  zero : Elem Nat ;                       -- zero constructor
  Even : Elem Nat -> Prop ;               -- evenness predicate
  EqNat : (m,n : Elem Nat) -> Prop ;      -- equality predicate
  prod : (m,n : Elem Nat) -> Elem Nat ; -- product function
```

To linearize the semantic categories and functions of the application grammar, we use grammatical categories and functions from the resource grammar:

```
lincat
   Dom = N ; -- Common Noun category
   Prop = S ; -- Sentence category
   Elem = NP ; -- Noun Phrase category
lin
```

```
zero = DefOneNP (UseN nol) ;
Even = predA1 (AdjP1 (adj1Star "чётн"));
EqNat = predV2 ravnjatsja ;
prod = appFunColl (funGen proizvedenie) ;
```

Some of the functions—`nol`, `ravnjatsja`, and `proizvedenie`—are lexical entities defined in the resource, ready with their inflectional forms ((which can mean dozens of forms in Russian), gender, etc. The application grammarian just has to pick the right ones. Some other functions—`adj1Star`—are lexical inflection patterns. To use them, one has to provide the word stem and choose the correct pattern.

The rest of the functions are from the language-independent API. Here are their type signatures:

```
AdjP1 : Adj1 -> AP ;                    -- adjective from lexicon
predA1 : AP -> VP ;                     -- adjectival predication
DefOneNP : CN -> NP ;                   -- singular definite phrase
UseN : N -> CN ;                        -- noun from lexicon
appFamColl : Fun -> NP -> NP -> NP ; -- collective function appl
predV2 : V2 -> NP -> NP -> NP -> S ; -- two-place verb predic
```

The user of the library has to be familiar with notions of constituency, but not with linguistic details such as inflection, agreement, and word order.

Writing even a small grammar in inflectionally rich language like Russian requires a lot of work on morphology. This is the part where using the resource grammar library really helps to speed up, since the resource functions for adding new lexical entries are relatively easy to use.

Syntactic rules are more tricky and require fair knowledge of the type system used. However, they heighten the level of the code written by using only function application. The resource style is also less error prone, since the correctness of the library functions is presupposed.

Using the resource grammar API, an application grammar can be implemented for different languages in a similar manner, since there is a shared language-independent API part and also because the libraries for different languages have similar structures. Often the same API functions can be used in different languages; but it may also happen that e.g. adjectival predication in one language is replaced by verbal predication in another.

Fig. 2 shows a simple theorem proof constructed by using the arithmetic grammars for Russian and English. The example was built with help of GF Syntax Editor [1].

3 Conclusion

A library of resource grammars is essential for a wider use of GF. In a grammar formalism, libraries are even more important than in a general-purpose programming language, since writing grammars for natural languages is such a

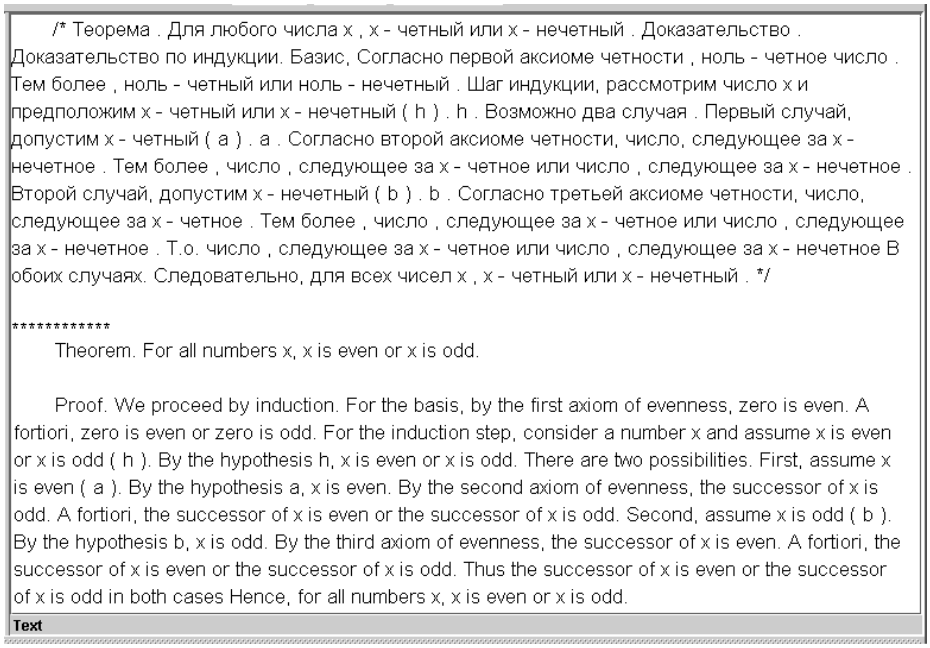

Fig. 2. Example of a theorem proof constructed using arithmetic grammars in Russian and English.

special kind of programming: it is easier to find a programmer who knows how to write a sorting algorithm than one who knows how to write a grammar for Russian relative clauses. To make GF widely used outside the latter group of programmers, resource grammars have to be created. Experience has shown that resource grammars for seemingly very different languages can share an API by which different grammars can be accessed in the same way. As a part of future work on the resource libraries, it remains to see how much divergent extensions of the common API are needed for different languages.

References

1. J. Khegai, B. Nordström, and A. Ranta. Multilingual syntax editing in GF. In A. Gelbukh, editor, *CICLing-2003, Mexico City, Mexico*, LNCS, pages 453–464. Springer, 2003.
2. M.Rayner, D.Carter, P.Bouillon, V.Digalakis, and M.Wirén. *The spoken language translator*. Cambridge University Press, 2000.
3. I.M. Pulkina. *A Short Russian Reference Grammar*. Russky Yazyk, Moscow, 1984.
4. A. Ranta. The GF Resource grammar library, 2002. `http://tournesol.cs.chalmers.se/aarne/GF/resource/`.
5. A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, to appear.
6. M.A. Shelyakin. *Spravochnik po russkoj grammatike (in Russian)*. Russky Yazyk, Moscow, 2000.
7. T. Wade. *A Comprehensive Russian Grammar*. Blackwell Publishing, 2000.

Paper 3.
Grammatical Framework (GF) for MT in Sublanguage Domains

Janna Khegai. "Grammatical Framework (GF) for MT in Sublanguage Domains". In proceedings of EAMT-2006 (11th Annual conference of the European Association for Machine Translation), pages 95-104, Oslo, Norway, 2006.

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-

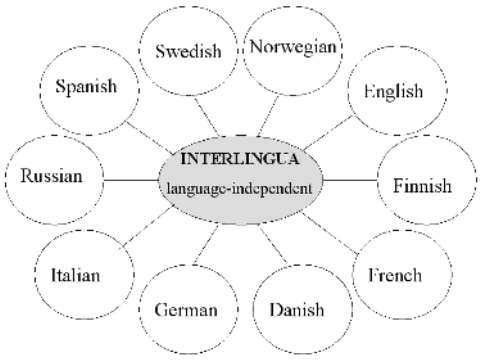

Figure 1: GF performs MT of interlingua type. Ten languages have general-purpose resource grammars that conform to a common interface.

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

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

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-

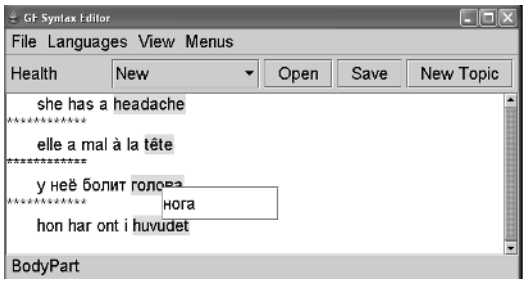

Figure 5: Editing text in Russian: a middle-click on a chosen word pop-ups a text filed, which can be used for replacing the current body part – *голова* (*head*) by a new one – *нога* (*leg*).

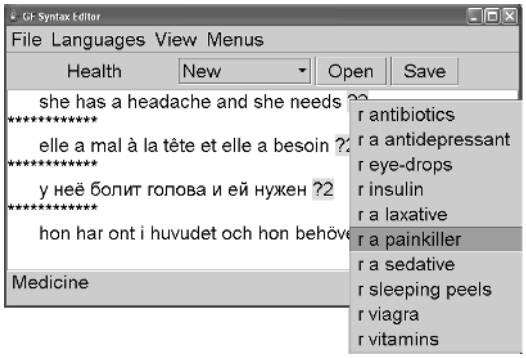

Figure 6: An editing menu in English is invoked by right-clicking on a placeholder (denoted by a question mark) in the sentence in English. The menu is generated automatically from the grammar.

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

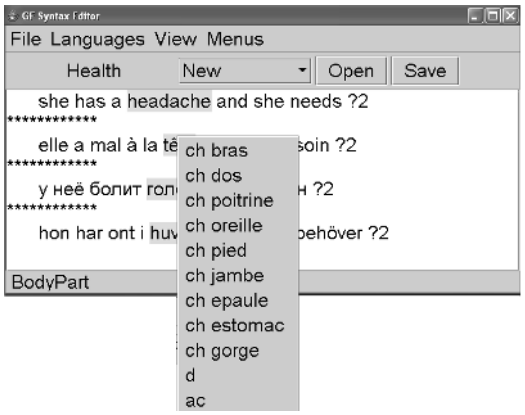

Figure 7: To get a context-dependent editing menu in French just right-click on the word in the French version.

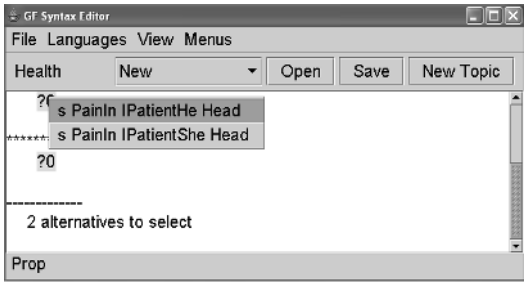

Figure 8: In case of ambiguity the system asks to choose among the available options. Here, after typing *I have a headache* the gender of the sentence's subject is required. In English the gender of the subject is not important for forming a correct sentence. However, the gender distinction is kept 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

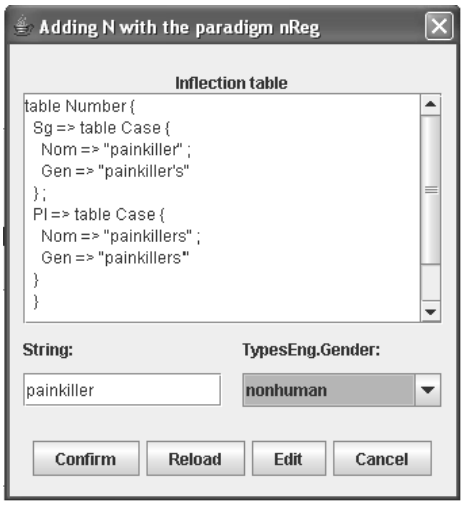

Figure 9: The GF IDE dialog window for adding lexicon entries. As indicated by the window caption, the inflection table has been generated from the stem string `painkiller` by using the `nReg` (`Reg`ular `n`oun) inflection pattern. We can see that all declension forms (by number: `Sg`, `Pl` and case: `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.

Paper 4.
Multilingual Syntax Editing in GF

Janna Khegai, Bengt Nordström and Aarne Ranta. "Multilingual Syntax Editing in GF". In Intelligent Text Processing and Computational Linguistics (CICLing-2003), Mexico. LNCS 2588, pages 453-464. Springer, 2003.

Multilingual Syntax Editing in GF

Janna Khegai, Bengt Nordström, and Aarne Ranta

Department of Computing Science
Chalmers University of Technology and Gothenburg University
SE-41296, Gothenburg, Sweden
{janna, bengt, aarne}@cs.chalmers.se

Abstract. GF (Grammatical Framework) makes it possible to perform multilingual authoring of documents in restricted languages. The idea is to use an object in type theory to describe the common abstract syntax of a document and then map this object to a concrete syntax in the different languages using linearization functions, one for each language. Incomplete documents are represented using metavariables in type theory. The system belongs to the tradition of logical frameworks in computer science. The paper gives a description of how a user can use the editor to build a document in several languages and also shows some examples how ambiguity is resolved using type checking. There is a brief description of how GF grammars are written for new domains and how linearization functions are defined.

1 Introduction

1.1 Multilingual authoring

We are interested in the problem of editing a document in several languages simultaneously. In order for the problem to be feasible, we use a restricted language. The idea is to use a mathematical structure (an object in type theory) as the basic representation of the document being edited. This structure describes the abstract syntax of the document. Concrete representations of the document in the various languages are expressed using linearization functions, one function for each language. The process of producing a concrete representation from the abstract object is thus deterministic, each abstract object has only one concrete representation in each language. The reverse problem (parsing) is not deterministic, a given concrete representation may correspond to many abstract objects (ambiguity). The way we resolve ambiguity is by having an interactive system, an ambiguity results in an incomplete abstract object which has to be completed by the user. For instance, in the phrase *Dear friend* it is not clear whether the friend is male of female, thus making a translation into Swedish impossible. In the corresponding abstract object there is a field for gender which has to be filled in by the user before the editing is complete.

Type theory is a completely formal language for mathematics developed by Martin-Löf in the 70's [9]. Versions of it are extensively used under the title of Logical Frameworks in various implementations of proof editors like Coq [19],

Alf [8], and Lego [7]. The type system of type theory is not only used to express syntactic well-formedness, but also semantic well-formedness. This means that a syntactically well-formed term also has a meaning. Moreover, it is often possible to use type checking to resolve ambiguities that a weaker grammatical description cannot resolve.

1.2 The GF Niche - Meaning-Based Technique

Grammatical Framework (GF) is a grammar formalism built upon a Logical Framework (LF). What GF adds to LF is a possibility to define concrete syntax, that is, notations expressing formal concepts in user-readable ways. The concrete syntax mechanism of GF is powerful enough to describe natural languages: like PATR [18] and HPSG [13], GF uses features and records to express complex linguistic objects. Although GF grammars are bidirectional like PATR and HPSG, the perspective of GF is on generation rather than parsing. This implies that grammars are built in a slightly different way, and also that generation is efficient enough to be performed in real time in interactive systems. Another difference from usual grammar formalisms is the support for multilinguality: it is possible to define several concrete syntaxes upon one abstract syntax. The abstract syntax then works as an interlingua between the concrete syntaxes. The development of GF as an authoring system started as a plug-in to the proof editor ALF, to permit natural-language rendering of formal proofs [5]. The extension of the scope outside mathematics was made in the Multilingual Document Authoring project at Xerox [3]. In continued work, GF has been used in areas like software specifications [4], instruction texts [6], and dialogue systems [17]. In general, GF works for any domain of language that permits a formal grammar. Since LF is more general than specific logical calculi, it is more flexible to use on different domains than, for instance, predicate calculus.

1.3 The Scope of the Paper

The GF program implementing the GF grammar formalism is a complex system able to perform many NLP tasks. For example, it can do the morphological analysis of French verbs, construct a letter in several languages, and even greet you in the morning using a speech synthesizer. In this paper, however, we choose to restrict the topic and only look at GF as a multilingual authoring tool. For a more elaborated description of the system, we refer to [16, 15].

The GF users can be divided into three competence levels:

- Author level
- Grammarian level
- Implementor level

On the author level all we can do is to work with pre-existing grammars. This level is described in Section 2. On the grammarian level we write grammars describing new language fragments. This, of course, requires acquaintance with

the GF formalism. Examples of work on this level are given in Section 3. On both of these levels, we have some control, for example, over parsing algorithms to be used. However, the full control of parsing, linearization, graphics and other algorithms, is only accessible on the implementor level. Since GF is open source software, any user who wants can also become an implementor; but describing the implementor level is outside the scope of this paper.

2 The GF Syntax Editor

The graphical user interface implemented in the GF Syntax Editor hides the complexity of the system from the naive user. It provides access to the system functionality without requiring knowledge of the GF formalism. In this section we will show a simple example of GF syntax editing procedure.

When you start the GF editor you choose the topic and the languages you want to work with. For instance, we decide to work within the LETTER topic and want to have translations in four languages: English, Swedish, French and Finnish. You can create a new editing object by choosing a category from the New list. For example, to construct a letter, choose the Letter category (Fig. 1). In Fig. 2 you can see the created object in the tree form in the left upper part as well as linearizations in the right upper part. The tree representation corresponds to the GF language-independent semantic representation, the GF abstract syntax or interlingua. The linearizations area displays the result of translation of abstract syntax representation into the corresponding language using the GF concrete syntax.

Fig. 1. The New menu shows the list of available categories within the current topic LETTER. Choosing the category Letter in the list will create an object of the corresponding type. The linearizations area contains a welcome message when the GF Editor has just been started.

According to the LETTER grammar a letter consists of a Heading, a Message and an Ending, which is reflected in the tree and linearizations structures.

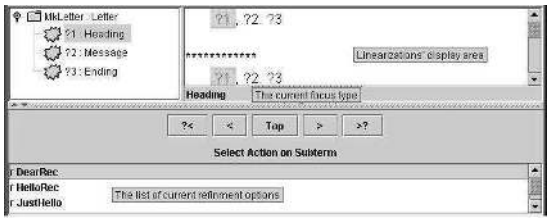

Fig. 2. The Abstract Syntax tree represents the letter structure. The current editing focus, the metavariable ?1 is highlighted. The type of the current focus is shown below the linearizations area. The context-dependent refinement option list is shown in the bottom part.

However, the exact contents of each of these parts are not yet known. Thus, we can only see question marks, representing metavariables, instead of language phrases in the linearizations.

Editing is a process of step-wise refinement, i.e. replacement of metavariables with language constructions. In order to proceed you can choose among the options shown in the refinement list. The refinement list is context-dependent, i.e. it refers to the currently selected focus. For example, if the focus is Heading, then we can choose among four options. Let us start our letter with the DearRec structure (Fig. 3(a)).

Now we have a new focus - metavariable ?4 of the type Recipient and a new set of refinement options. We have to decide what kind of recipient the letter has. Notice that the word *Dear* in Swedish and French versions is by default in male gender and, therefore, uses the corresponding adjective form. Suppose we want to address the letter to a female colleague. Then we choose the ColleagueShe option (Fig. 3(b)).

Notice that the Swedish and French linearizations now contain the female form of the adjective *Dear* , since we chose to write to a female recipient. This refinement step allows us to avoid the ambiguity while translating from English to, for example, a Swedish version of the letter.

Proceeding in the same fashion we eventually fill all the metavariables and get a completed letter like the one shown in Fig. 4(a).

A completed letter can be modified by replacing parts of it. For instance, we would like to address our letter to several male colleagues instead. We need first to move the focus to the Header node in the tree and delete the old refinement. In Fig. 5(a), we continue from this point by using the Read button, which invokes an input dialog, and expects a string to parse. Let us type *colleagues*.

The parsed string was ambiguous, therefore, as shown in Fig. 5(b), GF asks further questions. Notice that after choosing the ColleaguesHe option, not only the word *colleague* , but the whole letter switches to the plural, male form, see Fig. 4(b). In the English version only the noun *fellow* turns into plural, while in the other languages the transformations are more dramatic. The pronoun *you* turns into plural number. The participle *promoted* changes the number in the Swedish and French versions. The latter also changes the form of the verb *have*.

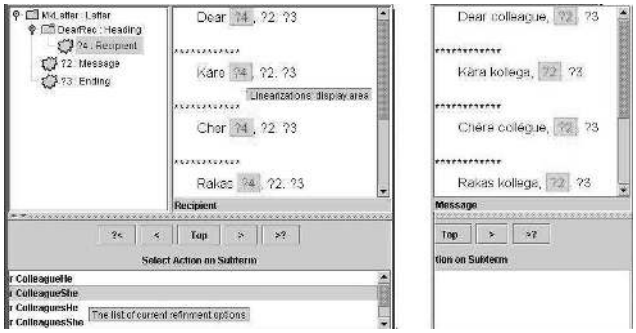

Fig. 3. (a) The linearizations are now filled with the first word that corresponds to *Dear* expression in English, Swedish, French and Finnish. The refinement focus is moved to the Recipient metavariable. (b) The Heading part is now complete. The adjective form changes to the corresponding gender after choosing the recipient.

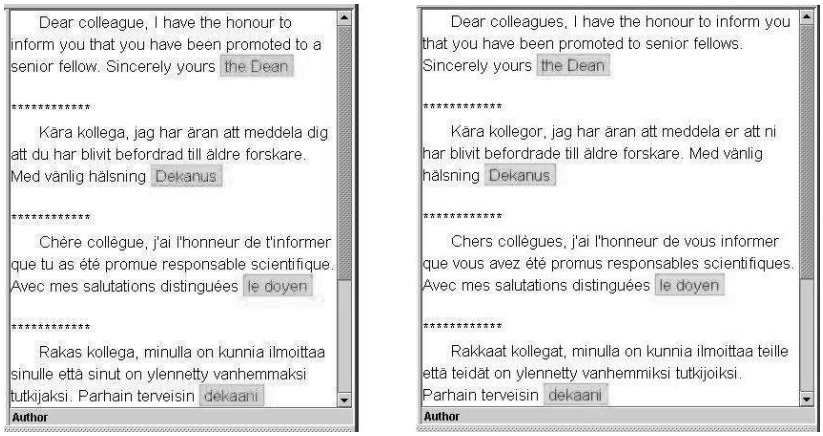

Fig. 4. (a) The complete letter in four languages. (b) Choosing the plural male form of the Recipient causes various linguistic changes in the letter as compared to (a).

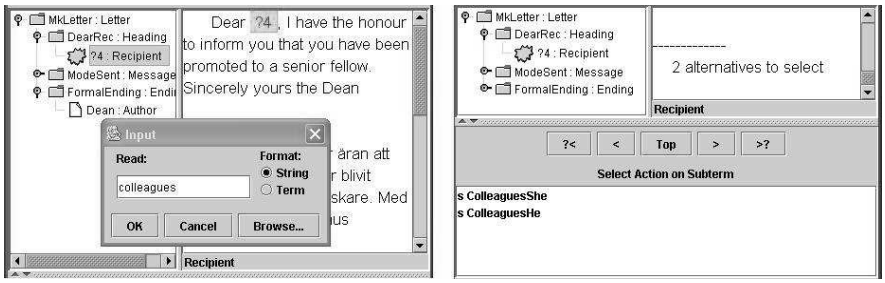

Fig. 5. (a) A refinement step can be done by using the Read button, which asks the user for a string to parse. (b) When the parsed string in (a) is ambiguous GF presents two alternative ways to resolve the ambiguity.

Both the gender and the number affect the adjective *dear* in French, but only the number changes in the corresponding Finnish adjective. Thus, the refinement step has led to substantial linguistic changes.

3 The GF grammar formalism

The syntax editor provided by GF is generic with respect to both subject matters and target languages. To create a new subject matter (or modify an old one), one has to create (or edit) an *abstract syntax*. To create a new target language, one has to work on a *concrete syntax*. Target languages can be added on the fly: if a new language is selected from the Language menu, a new view appears in the editor while other things remain equal, including the document that is being edited. Fig. 6 shows the effect of adding Russian to the above example.

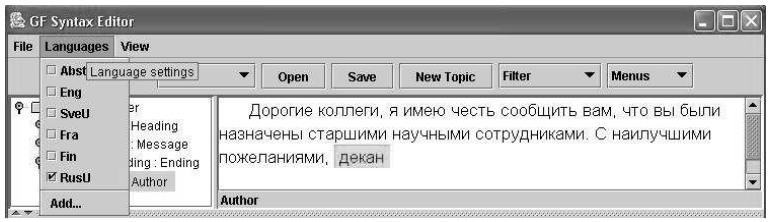

Fig. 6. Now we are able to translate the letter into Russian

The syntax editor itself is meant to be usable by people who do not know GF, but just something of the subject matter and at least one of the target languages. Authoring GF grammars requires expertise on both the subject matter and the target languages, and of course some knowledge of the GF grammar formalism.

A typical GF grammar has an abstract syntax of 1–3 pages of GF code, and concrete syntaxes about the double of that size. The use of resource grammars (Section 3.5) makes concrete syntaxes much shorter and easier to write.

3.1 Abstract syntax: simple example

An abstract syntax gives a structural description of a domain. It can be semantically as detailed and rigorous as a mathematical theory in a Logical Framework. It can also be less detailed, depending on how much semantic control of the document is desired. Asq an example, consider a fragment of the abstract syntax for letters:

```
cat Letter ; Recipient ; Author ; Message ;
    Heading ; Ending ; Sentence ;

fun MkLetter : Heading -> Message -> Ending -> Letter ;
fun DearRec : Recipient -> Heading ;
```

```
fun PlainSent : Sentence -> Message ;
fun ColleagueHe, ColleaguesShe : Recipient ;
```

The grammar has a set of categories `cat` and functions `fun`. The functions are used for building abstract syntax trees, and each tree belongs to a category. When editing proceeds, the system uses the abstract syntax to build a menu of possible actions: for instance, the possible refinements for a metavariable of type C are those functions whose value type is C.

3.2 Concrete syntax

Concrete syntax maps abstract syntax trees into linguistic objects. The objects can be simply strings, but in general they are records containing inflection tables, agreement features, etc. Each record has a type, which depends on the category of the tree, and of course on the target language. For instance, a part of the English category system for letters is defined as follows:

```
param Num = Sg | Pl ;
param Agr = Ag Num ;

lincat Letter = {s : Str} ;
lincat Recipient, Author = {s : Str ; a : Agr} ;
lincat Message = {s : Agr => Agr => Str} ;
lincat Heading, Ending = {s : Str ; a : Agr} ;
```

Both the author and the recipient have inherent agreement features (number), which are passed from them to the message body, so that right forms of verbs and nouns can be selected there:

```
lin MkLetter head mess end = {s =
      head.s ++ "," ++
      mess.s ! end.a ! head.a ++ "." ++
      end.s} ;
```

Different languages have different parameter systems. French, for instance, has gender in addition to number in the agreement features:

```
param Gen = Masc | Fem ; Num = Sg | Pl ; Agr = Ag Gen Num ;
```

3.3 Semantic control in abstract syntax

The main semantic control in the letter grammar is the structure that is imposed on all letters: it is, for instance, not possible to finish a letter without a heading. This kind of control would be easy to implement even using a context-free grammar or XML. Semantic control of more demanding kind is achieved by using the *dependent types* of type theory. For instance, the abstract syntax

```
cat Text ; Prop ; Proof (A : Prop) ;
fun ThmWithProof, ThmHideProof : (A : Prop) -> Proof A -> Text ;
```

defines mathematical texts consisting of a proposition and a proof. The type of proofs depends on propositions: the type checker can effectively decide whether a given proof really is a proof of a given theorem. Type checking also helps the author of the proof by only showing menu items that can possibly lead to a correct proof. Proof texts are linearized by to the following rules:

```
lin ThmWithProof A P =
      {s = "Theorem." ++ A.s ++ "Proof." ++ P.s ++ "Q.E.D."} ;
lin ThmHideProof A P =
      {s = "Theorem." ++ A.s ++ "Proof." ++ "Omitted."} ;
```

The latter form omits the proof, but the author is nevertheless obliged to construct the proof in the internal representation.

Mathematical texts with hidden proofs are a special case of *proof-carrying documents*, where semantic conditions are imposed by using dependent types in abstract syntax (cf. the notion of *proof-carrying code* [12]). Consider texts describing flight connections:

> *To get from Gothenburg to New York, you can first fly SK433 to Copenhagen and then take SK909.*

There are three conditions: that SK433 flies from Gothenburg to Copenhagen, that SK909 flies from Copenhagen to New York, and that change in Copenhagen is possible. These conditions are expressed by the following abstract syntax:

```
cat City ; Flight (x,y : City) ;

fun Connection :
      (x,y,z : City) -> (a : Flight x y) -> (b : Flight y z)
            -> Proof (PossibleChange x y z a b) -> Flight x z ;
fun PossibleChange :
      (x,y,z : City) -> Flight x y -> Flight y z -> Prop ;
```

The linearization rule for `Connection` produces texts like the example above, with internal representation that includes the hidden proof. We have left it open how exactly to construct proofs that a change is possible between two flights: this involves a proof that the departure time of the second flight lies within a certain interval from the arrival time of the first flight, the minimum length of the interval depending on the cities involved. In the end, the proof condition reduces to ordinary mathematical concepts.

3.4 Semantic disambiguation

An important application of semantic control is *disambiguation*. For instance, the English sentence

> *there exists an integer x such that x is even and x is prime*

has two French translations,

il existe un entier x tel que x soit pair et que x soit premier
il existe un entier x tel que x soit pair et x est premier

corresponding to the trees

```
Exist Int (\x -> Conj (Even x) (Prime x))
Conj (Exist Int (\x -> Even x)) (Prime x)
```

respectively. Both analyses are possible by context-free parsing, but type checking rejects the latter one because it has an unbound occurrence of x.

Another example of semantic disambiguation is the resolution of pronominal reference. The English sentence

if the function f has a maximum then it reaches it at 0

has two occurrences of *it*. Yet the sentence is not ambiguous, since it uses the predicate *reach*, which can only take the function as its first argument and the maximum as its second argument: the dependently typed syntax tree uses a pronominalization function

```
fun Pron : (A : Dom) -> Elem A -> Elem A
```

making the domain and the reference of the pronoun explicit. Linearization rules of `Pron` into languages like French and German use the domain argument to select the gender of the pronoun, so that, for instance, the German translation of the example sentence uses *sie* for the first *it* and *es* for the second:

wenn die Funktion f ein Maximum hat, dann reicht sie es bei 0

3.5 Application grammars and resource grammars

GF is primarily geared for writing specialized grammars for specialized domains. It is possible to avoid many linguistic problems just by ignoring them. For instance, if the grammar only uses the present tense, large parts of verb conjugation can be ignored. However, always writing such grammars from scratch has several disadvantages. First, it favours solutions that are linguistically *ad hoc*. Secondly, it produces concrete syntaxes that are not reusable from one application to another. Thirdly, it requires the grammarian simultaneously to think about the domain and about linguistic facts such as inflection, agreement, word order, etc. A solution is to raise the level of abstraction, exploiting the fact that GF is a functional programming language: do not define the concrete syntax as direct mappings from trees to strings, but as mappings from trees to structures in a *resource grammar*.

A resource grammar is a generic description of a language, aiming at completeness. From the programming point of view, it is like a *library module*, whose proper use is via type signatures: the user need not know the definitions of module functions, but only their types. This modularity permits a division of labour between programmers with different expertises. In grammar programming, there is typically a domain expert, who knows the abstract syntax and wants to map it

into a concrete language, and a linguist, who has written the resource grammar and provided a high-level interface to it.

For instance, the resource grammar may contain linguistic categories, syntactic rules, and (as a limiting case of syntactic rules) lexical entries:

```
cat S ; NP ; Adj ;              -- sentence, noun phrase, adjective
fun PredAdj : Adj -> NP -> S ; -- "NP is Adj"
fun Condit : S -> S -> S ;     -- "if S then S"
fun adj_even : Adj ;           -- "even"
```

The author of a grammar of arithmetic proofs may have the following abstract syntax with semantically motivated categories and functions:

```
cat Prop ; Nat ;                 -- proposition, natural number
fun If : Prop -> Prop -> Prop ; -- logical implication
fun Ev : Nat -> Prop ;           -- the evenness predicate
```

The concrete syntax that she writes can exploit the resource grammar:

```
lincat Prop = S ; Nat = NP ;
lin If = Condit ;
lin Ev = PredAdj adj_even ;
```

Experience with GF has shown that the abstract interfaces to resource grammars can largely be shared between different languages. Thus a German resource grammar can have the same type signatures as the English one, with the exception of lexical rules. In this case we have

```
lin Ev = PredAdj adj_gerade ;
```

Yet the effects of these rules are language-dependent. German has more agreement and word order variation in the conditional and predication rules. For instance, the syntax tree `If (Ev n1) (Od n3)` is linearized as follows:

English: *if* 1 *is even then* 2 *is odd*
German: *wenn* 1 *gerade ist, dann ist* 2 *ungerade*

Of course, there are also cases where different linguistic structures must be used in the abstract syntax. For instance, the two-place predicate saying that x misses y is expressed by a two-place verb construction in both English and French, but the roles of subject and object are inverted (x *misses* y vs. y *manque à* x):

English: `lin Miss x y = PredVP x (ComplV2 verb_miss y)`
French: `lin Miss x y = PredVP y (ComplV2 verb_manquer x)`

4 Discussion

4.1 Comparison to other systems

In computer science, one of the earliest attempt of generating a syntax editor from a language description was the Mentor [2] system at INRIA. Another early

example is the Cornell program synthesizer [20], which uses an attribute grammar formalism to describe the language.

The idea of using a strictly formalized language for mathematics to express the abstract syntax of natural language was proposed by Curry [1] and used by Montague [11] in his denotational semantics of English. The followers of Montague, however, usually ignore the abstract syntax and define relations between natural language and logic directly. This makes generation much harder than when using an abstract syntax tree as the primary representation.

The WYSIWYM system [14] by Power and Scott has many similarities with our system. It is also a system for interactive multi-lingual editing. WYSIWYM does not use a mathematical language to express abstract syntax and it seems not to be possible for the user to change the structure of what they call the knowledge base (our abstract syntax).

Processing natural language within restricted domains makes GF related to the KANT translation system [10]. Kant Controlled English (KCE) put constraints on vocabulary, grammar and document structure in order to reduce the amount of ambiguity in the source text in the pre-processing phase. The remaining ambiguities are resolved via interaction with the author in the authoring environment. The only source language in the KANT system is English. KANT does not use any formal semantic representation.

4.2 Future work

We would like to see the system as a (structured) document editor. This has many implications. The major part of the screen will in the future be devoted to the different documents and not to the various menus. The parts of the document which is not filled in – now indicated by metavariables – will have a meaningful label expressed in the language being presented. The natural action of refining a metavariable by parsing a text from the keyboard is to put the cursor on the metavariable and start typing. In the future it will also be possible to use completions, so when the user enters for instance a tab character the system responds with the longest unique possible continuation of the input together with a list of alternatives. Completion in GF can be controlled both by the application grammar and by statistics of the interaction history.

A grammarian-level user needs an advanced editor for editing grammar. So a natural idea is to extend GF to make it possible to edit the GF formalism itself.

Creating resource grammars is an important part in the development of GF, corresponding to the development of standard libraries for programming languages. The current libraries (end 2002) contain basic morphology, phrase structure, and agreement rules for English, French, German, and Swedish.

GF is generic program capable of using any set of grammars in an. Hardwired grammars, however, permit more light-weight implementations. We use the name *gramlets* for Java programs implementing such light-weight special-purpose editors. Gramlets can be used in PDAs and as web applets, and they can be automatically compiled from GF grammars.

References

1. H. B. Curry. Some logical aspects of grammatical structure. In Roman Jakobson, editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Twelfth Symposium in Applied Mathematics*, pages 56–68. American Mathematical Society, 1963.
2. V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*, 1975.
3. M. Dymetman, V. Lux, and A. Ranta. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249, 2000.
4. R. Hähnle, K. Johannisson, and A. Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.
5. T. Hallgren and A. Ranta. An extensible proof text editor. In M. Parigot and A. Voronkov, editors, *LPAR-2000*, volume 1955 of *LNCS/LNAI*, pages 70–84. Springer, 2000.
6. K. Johannisson and A. Ranta. Formal verification of multilingual instructions. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2001.
7. Z. Luo and R. Pollack. LEGO Proof Development System. Technical report, University of Edinburgh, 1992.
8. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS 806, pages 213–237. Springer, 1994.
9. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.
10. T. Mitamura and E. H. Nyberg. Controlled English for Knowledge-Based MT: Experience with the KANT system. In *TMI*, 1995.
11. R. Montague. *Formal Philosophy*. Yale University Press, New Haven, 1974. Collected papers edited by R. Thomason.
12. G. C. Necula. Proof-Carrying Code. In *Proc. 24th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 106–119. ACM Press, 1997.
13. C. Pollard and I. Sag. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, 1994.
14. R. Power and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL*, 1998.
15. A. Ranta. GF Homepage, 2002. `www.cs.chalmers.se/~aarne/GF/`.
16. A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, to appear.
17. A. Ranta and R. Cooper. Dialogue systems as proof editors. In *IJCAR/ICoS-3*, Siena, Italy, 2001.
18. S. Shieber. *An Introduction to Unification-Based Approaches to Grammars*. University of Chicago Press, 1986.
19. Coq Development Team. Coq Homepage. `http://pauillac.inria.fr/coq/`, 1999.
20. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

Technical report A

GF Russian resource library[5]

One of the features of GF is separation between the language description (grammars) and the processing engine. Grammars are written using the GF language and stored in text files. Therefore, grammars can be considered as programs written in the GF grammar language, which can be compiled and run by GF system. Just as with ordinary programming languages the efficiency of programming labor can be significantly increased by reusing previously written code. For that purpose standard libraries are usually used.

To use the library a programmer only needs to know the type signatures of the library functions. Implementation details are usually hidden from the user. The GF resource grammar library [31] is aimed to serve as a standard library for the GF grammar language. Since GF is a multlingual system the library structure has an additional dimension for different languages. Each language has its own layer and all layers have more or less similar internal structure. Some parts of the library are language-independent and shared among the languages.

The implementation of Russian resource grammar proves that GF grammar formalism allows us to use the language-independent abstract API for describing sometimes rather peculiar grammatical variations in different languages.

A resource grammar has 13 language-specific modules (not including lexicon), see Fig. 2 that contain implementations for about 125 language-independent API-rules (not including paradigms). A standard lexicon included for a language has 450 words.

The first section of this chapter is an overview of syntactic structures borrowed from the resource library documentation written by Aarne Ranta. Resource module contains the description of the basic parameters and operations used in other modules. Categories module is an overview of the grammar library types. In the rest of the chapter the modules are listed alphabetically. The concluding sections contain descriptions of some Russian morphological paradigms and automatically generated test examples intended for proof-reading and also reflecting

[5]Written together with Aarne Ranta.

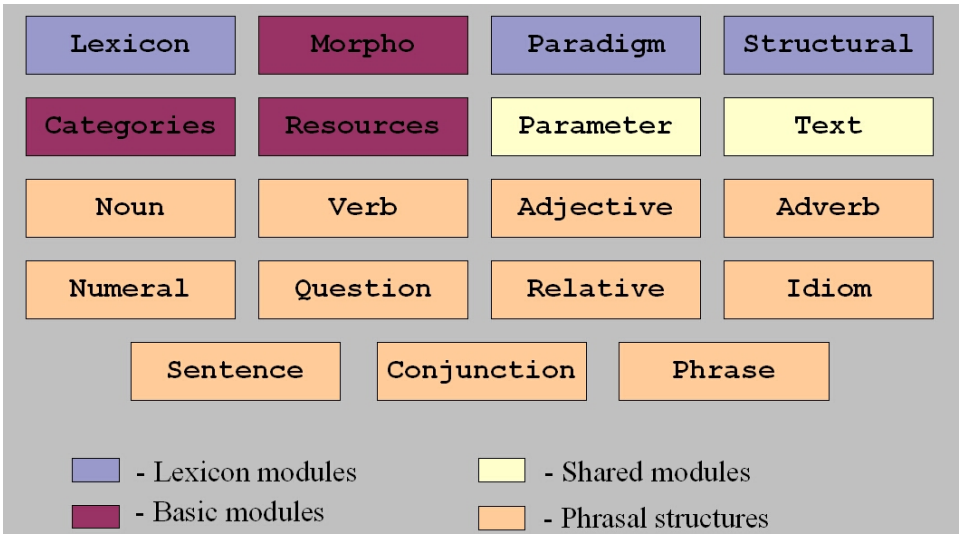

Figure 2: Resource grammar modules.

the coverage of the GF Russian resource library.

1 Overview of syntactic structures

1.1 Texts. phrases, and utterances

The outermost linguistic structure is `Text`. `Text`s are composed from Phrases (`Phr`) followed by punctuation marks - either of ".", "?" or "!" (with their proper variants in Spanish and Arabic). Here is an example of a `Text` string.

```
John walks. Why? He doesn't want to sleep!
```

Phrases are mostly built from Utterances (`Utt`), which in turn are declarative sentences, questions, or imperatives - but there are also "one-word utterances" consisting of noun phrases or other subsentential phrases. Some Phrases are atomic, for instance "yes" and "no". Here are some examples of Phrases.

```
yes
come on, John
but John walks
give me the stick please
don't you know that he is sleeping
a glass of wine
a glass of wine please
```

There is no connection between the punctuation marks and the types of utterances. This reflects the fact that the punctuation mark in a real text is selected

as a function of the speech act rather than the grammatical form of an utterance. The following text is thus well-formed.

```
John walks. John walks? John walks!
```

What is the difference between Phrase and Utterance? Just technical: a Phrase is an Utterance with an optional leading conjunction ("but") and an optional tailing vocative ("John", "please").

1.2 Sentences and clauses

The richest of the categories below Utterance is `S`, Sentence. A Sentence is formed from a Clause (`Cl`), by fixing its Tense, Anteriority, and Polarity. For example, each of the following strings has a distinct syntax tree in the category Sentence:

```
John walks
John doesn't walk
John walked
John didn't walk
John has walked
John hasn't walked
John will walk
John won't walk
...
```

whereas in the category Clause all of them are just different forms of the same tree. The difference between Sentence and Clause is thus also rather technical. It may not correspond exactly to any standard usage of the terms "sentence" and "clause".

Fig.3 shows a type-annotated syntax tree of the Text "John walks." and gives an overview of the structural levels.

Here are some examples of the results of changing constructors.

```
 1. TFullStop -> TQuestMark   John walks?
 3. NoPConj   -> but_PConj    But John walks.
 6. TPres     -> TPast        John walked.
 7. ASimul    -> AAnter       John has walked.
 8. PPos      -> PNeg         John doesn't walk.
11. john_PN   -> mary_PN      Mary walks.
13. walk_V    -> sleep_V      John sleeps.
14. NoVoc     -> please_Voc   John sleeps please.
```

All constructors cannot of course be changed so freely, because the resulting tree would not remain well-typed. Here are some changes involving many constructors:

```
Node Constructor             Value type  Other constructors
-------------------------------------------------------------
 1.  TFullStop               Text        TQuestMark
 2.    (PhrUtt               Phr
 3.      NoPConj             PConj       but_PConj
 4.      (UttS               Utt         UttQS
 5.        (UseCl            S           UseQCl
 6.           TPres          Tense       TPast
 7.           ASimul         Anter       AAnter
 8.           PPos           Pol         PNeg
 9.           (PredVP        Cl
10.             (UsePN       NP          UsePron, DetCN
11.               john_PN)   PN          mary_PN
12.             (UseV        VP          ComplV2, ComplV3
13.               walk_V)))) V           sleep_V
14.       NoVoc)             Voc         please_Voc
15.    TEmpty                Text
```

Figure 3. Type-annotated syntax tree of the Text "John walks."

```
 4- 5. UttS (UseCl ...) ->
         UttQS (UseQCl (... QuestCl ...)) Does John walk?
10-11. UsePN john_PN    ->
         UsePron we_Pron                   We walk.
12-13. UseV walk_V      ->
         ComplV2 love_V2 this_NP           John loves this.
```

1.3 Parts of sentences

The linguistic phenomena mostly discussed in both traditional grammars and modern syntax belong to the level of Clauses, that is, lines 9-13, and occasionally to Sentences, lines 5-13. At this level, the major categories are `NP` (Noun Phrase) and `VP` (Verb Phrase). A Clause typically consists of just an `NP` and a `VP`. The internal structure of both `NP` and `VP` can be very complex, and these categories are mutually recursive: not only can a `VP` contain an `NP`,

```
[VP loves [NP Mary]]
```

but also an `NP` can contain a `VP`

```
[NP every man [RS who [VP walks]]]
```

(a labelled bracketing like this is of course just a rough approximation of a GF syntax tree, but still a useful device of exposition).

Most of the resource modules thus define functions that are used inside NPs and VPs. Here is a brief overview:

Noun. How to construct NPs. The main three mechanisms for constructing NPs are

- from proper names: "John"
- from pronouns: "we"
- from common nouns by determiners: "this man"

The `Noun` module also defines the construction of common nouns. The most frequent ways are

- lexical noun items: "man"
- adjectival modification: "old man"
- relative clause modification: "man who sleeps"
- application of relational nouns: "successor of the number"

Verb. How to construct VPs. The main mechanism is verbs with their arguments, for instance,

- one-place verbs: "walks"
- two-place verbs: "loves Mary"
- three-place verbs: "gives her a kiss"
- sentence-complement verbs: "says that it is cold"
- VP-complement verbs: "wants to give her a kiss"

A special verb is the copula, "be" in English but not even realized by a verb in all languages. A copula can take different kinds of complement:

- an adjectival phrase: "(John is) old"
- an adverb: "(John is) here"
- a noun phrase: "(John is) a man"

Adjective. How to constuct `AP`s. The main ways are

- positive forms of adjectives: "old"
- comparative forms with object of comparison: "older than John"

Adverb. How to construct `Adv`s. The main ways are

- from adjectives: "slowly"
- as prepositional phrases: "in the car"

1.4 Modules and their names

The resource modules are named after the kind of phrases that are constructed in them, and they can be roughly classified by the "level" or "size" of expressions that are formed in them:

- Larger than sentence: `Text`, `Phrase`
- Same level as sentence: `Sentence`, `Question`, `Relative`
- Parts of sentence: `Adjective`, `Adverb`, `Noun`, `Verb`
- Cross-cut (coordination): `Conjunction`

Because of mutual recursion such as in embedded sentences, this classification is not a complete order. However, no mutual dependence is needed between the modules themselves - they can all be compiled separately. This is due to the module `Cat`, which defines the type system common to the other modules. For instance, the types `NP` and `VP` are defined in `Cat`, and the module `Verb` only needs to know what is given in `Cat`, not what is given in `Noun`. To implement a rule such as

```
Verb.ComplV2 : V2 -> NP -> VP
```

it is enough to know the linearization type of `NP` (as well as those of `V2` and `VP`, all given in `Cat`). It is not necessary to know what ways there are to build `NP`s (given in `Noun`), since all these ways must conform to the linearization type defined in `Cat`. Thus the format of category-specific modules is as follows:

```
abstract Adjective = Cat ** {...}
abstract Noun      = Cat ** {...}
abstract Verb      = Cat ** {...}
```

1.5 Top-level grammar and lexicon

The module `Grammar` collects all the category-specific modules into a complete grammar:

```
abstract Grammar =
  Adjective, Noun, Verb, ..., Structural, Idiom
```

The module `Structural` is a lexicon of structural words (function words), such as determiners.

The module `Idiom` is a collection of idiomatic structures whose implementation is very language-dependent. An example is existential structures ("there is", "es gibt", "il y a", etc).

The module `Lang` combines `Grammar` with a `Lexicon` of ca. 350 content words:

```
abstract Lang = Grammar, Lexicon
```

Using `Lang` instead of `Grammar` as a library may give for free some words needed in an application. But its main purpose is to help testing the resource library, rather than as a resource itself. It does not even seem realistic to develop a general-purpose multilingual resource lexicon.

1.6 Language-specific syntactic structures

The API collected in `Grammar` has been designed to be implementable for all languages in the resource package. It does contain some rules that are strange or superfluous in some languages; for instance, the distinction between definite and indefinite articles does not apply to Finnish and Russian. But such rules are still easy to implement: they only create some superfluous ambiguity in the languages in question.

But the library makes no claim that all languages should have exactly the same abstract syntax. The common API is therefore extended by language-dependent rules. The top level of each languages looks as follows (with English as example):

```
abstract English = Grammar, ExtraEngAbs, DictEngAbs
```

where `ExtraEngAbs` is a collection of syntactic structures specific to English, and `DictEngAbs` is an English dictionary (at the moment, it consists of `IrregEngAbs`, the irregular verbs of English). Each of these language-specific grammars has the potential to grow into a full-scale grammar of the language. These grammars can also be used as libraries, but the possibility of using functors is lost.

To give a better overview of language-specific structures, modules like are built (for example, `ExtraEngAbs`) from a language-independent module `ExtraAbs` by restricted inheritance:

```
abstract ExtraEngAbs = Extra [f,g,...]
```

Thus any category and function in `Extra` may be shared by a subset of all languages. One can see this set-up as a matrix, which tells what `Extra` structures are implemented in what languages. For the common API in `Grammar`, the matrix is filled with 1's (everything is implemented in every language).

Language-specific extensions and the use of restricted inheritance is a recent addition to the resource grammar library, and has only been exploited in a very small scale so far. `ExtraAbs` and `DictAbs` are not implemented for Russian.

2 Resource

.

This module contains operations that are needed to make the resource syntax work. To define everything that is needed to implement `Test`, it moreover contains regular lexical patterns needed for `Lex`.

```
resource ResRus = ParamX ** open Prelude in {
 flags  coding=utf8 ; optimize=all ;
```

The flag `coding=utf8` indicates that Russian grammars use UTF-8 encoding for Cyrillic letters. The flag `optimize=all` turns on the optimization procedure, which can result in reducing the grammar size up to 50% of the original.

2.1 Enumerated parameter types

These types are the ones found in school grammars. Their parameter values are atomic. Some parameters, such as `Number`, are inherited from `ParamX`.

Declination forms depend on Case, Animacy , Gender: *большие дома - больших домов* (*big houses - big houses'*); and on Number: *большой дом - большие дома* (*a big house - big houses*).

There are three genders: masculine, feminine and neuter. A number of Russian nouns have common gender. They can denote both males and females: *умница* (*a clever person*), *инженер* (*an engineer*). We overlook this phenomenon for now.

```
  param
    Gender      = Masc | Fem | Neut ;
```

There are six cases: nominative, genitive, dative, accusative, instructive and prepositional:

```
    Case        = Nom | Gen | Dat | Acc | Inst | Prepos ;
```

Animacy plays role only in the Accusative case (Masc Sg and Plural forms): Accusative Animate = Genitive, Accusaive Inanimate = Nominative *я люблю большие дома - я люблю больших мужчин* (*I love big houses - I love big men*):

```
Animacy    = Animate | Inanimate ;
```

There are two voices: active and passive:

```
Voice      = Act | Pass ;

Aspect     = Imperfective | Perfective ;
```

The AfterPrep parameter is introduced in order to describe the variations of the third person personal pronoun forms depending on whether they come after a preposition or not:

```
AfterPrep  = Yes | No ;
```

`Possessive` parameter is introduced for personal pronouns: *я - мой* (*I - mine*):

```
Possessive = NonPoss | Poss GenNum ;
```

This `ClForm` is introduced for the compatibility with the current language-independent API:

```
ClForm =  ClIndic Tense Anteriority | ClCondit  |
          ClInfinit | ClImper;
```

The plural never makes a gender distinction, for example, for adjectives:

```
GenNum = ASg Gender | APl ;
```

Coercions between the compound gen-num type and gender and number:

```
oper
  gNum : Gender -> Number -> GenNum = \g,n ->
    case n of
   {   Sg => case g of
                { Fem => ASg Fem ;
                  Masc => ASg Masc ;
                  Neut => ASg Neut  } ;
       Pl => APl
   } ;
```

The Possessive parameter is introduced in order to describe the possessives of personal pronouns, which are used in the Genitive constructions like *моя мама* (*my mother*) instead of *мама моя* (*the mother of mine*).

2.2 For `Noun`

Nouns decline according to number and case. For the sake of shorter description these parameters are combined in the type SubstForm.

```
param
  SubstForm = SF Number Case ;
```

Real parameter types (i.e. ones on which words and phrases depend) are mostly hierarchical. The alternative would be cross-products of simple parameters, but this would usually overgenerate. However, we use the cross-products in complex cases (for example, aspect and tense parameter in the verb description) where the relationship between the parameters are non-trivial even though we aware that some combinations do not exist (for example, present perfective does not exist, but removing this combination would lead to having different descriptions for perfective and imperfective verbs, which we do not want for the sake of uniformity).

```
param  PronForm = PF Case AfterPrep Possessive;

oper Pronoun = { s : PronForm => Str ; n : Number ; p : Person ;
           g: PronGen ;  pron: Bool} ;
```

Gender is not morphologically determined for first and second person pronouns.

```
param  PronGen = PGen Gender | PNoGen ;
```

The following coercion is useful:

```
oper
 pgen2gen : PronGen -> Gender = \p -> case p of {
  PGen g => g ;
  PNoGen => variants {Masc ; Fem}
  } ;

oper
  extCase: PronForm -> Case = \pf -> case pf of
   { PF Nom _ _ => Nom ;
     PF Gen _ _ => Gen ;
     PF Dat _ _ => Dat ;
     PF Inst _ _ => Inst ;
     PF Acc _ _ => Acc ;
     PF Prepos _ _ => Prepos
    } ;
```

```
mkPronForm: Case -> AfterPrep -> Possessive -> PronForm =
    \c,n,p -> PF c n p ;

CommNounPhrase: Type = {s : Number => Case => Str;
                        g : Gender; anim : Animacy} ;

NounPhrase : Type = { s : PronForm => Str ; n : Number ;
   p : Person ; g: PronGen ; anim : Animacy ;  pron: Bool} ;

mkNP : Number -> CommNounPhrase -> NounPhrase = \n,chelovek ->
    {s = \\cas => chelovek.s ! n ! (extCase cas) ;
     n = n ; g = PGen chelovek.g ; p = P3 ; pron =False ;
     anim = chelovek.anim
    } ;

det2NounPhrase : Adjective -> NounPhrase = \eto ->
    {s = \\pf => eto.s ! (AF (extCase pf) Inanimate (ASg Neut));
     n = Sg ; g = PGen Neut ; pron = False ; p = P3 ;
     anim = Inanimate } ;

pron2NounPhraseNum : Pronoun -> Animacy -> Number -> NounPhrase =
   \ona, anim, num ->
    {s = ona.s ; n = num ; g =  ona.g ;
     pron = ona.pron; p = ona.p ; anim = anim } ;
```

Agreement of `NP` is a record. We'll add `Gender` later.

```
oper  Agr = {n : Number ; p : Person} ;
```

2.3 For `Verb`

`Mood` is the main verb classification parameter. The verb mood can be infinitive, subjunctive, imperative, and indicative. Note: subjunctive mood is analytical, i.e. formed from the past form of the indicative mood plus the particle *чтобы*. That is why they have the same `GenNum` parameter. We choose to keep the "redundant" form in order to indicate the presence of the subjunctive mood in Russian verbs.

`Aspect` and `Voice` parameters are present in every mood, so `Voice` is put before the mood parameter in the verb form description hierachy. Moreover `Aspect` is regarded as an inherent parameter of a verb entry. The primary reason for that is that one imperfective form can have several perfective forms: *ломать* - *сломать* - *поломать* (*to break*). Besides, the perfective form could be formed from imperfective by prefixation, but also by taking a completely different stem: *говорить-сказать* (*to say*). In the later case it is even natural to regard them

as different verb entries. Another reason is that looking at the `Aspect` as an inherent verb parameter seem to be customary in other similar projects, see [34].

Note: Of course, the whole inflection table has many redundancies in the sense that many verbs do not have all grammatically possible forms. For example, passive does not exist for the verb *любить* (*to love*), but exists for the verb *ломать* (*to break*). In present tense verbs do not conjugate according to `Gender`, so parameter `GenNum` is used instead of `Number`, for the sake of using a verb, for example, as an adjective in predication. Depending on the tense verbs conjugate according to combinations of gender, person and number of the verb objects. Participles (present and past) and gerund forms are not included in the current description. This is the verb type used in the lexicon:

```
oper Verbum : Type = { s: VerbForm => Str ; asp : Aspect };

param

  VerbForm = VFORM Voice VerbConj ;
  VerbConj =  VIND GenNum VTense | VIMP Number Person |
              VINF | VSUB GenNum ;
  VTense   = VPresent Person | VPast | VFuture Person ;

oper
 getVTense : Tense -> Person -> VTense= \t,p -> case t of
 {Present => VPresent p ; Past => VPast; Future => VFuture p } ;

 getVoice: VerbForm -> Voice = \vf ->
   case vf of {
    VFORM Act _ => Act;
    VFORM Pass _ => Pass
 };

 sebya : Case => Str =table {
     Nom => "";

    Gen => "себя";
    Dat=> "себе";
    Acc => "себя";
    Instr => "собой";
    Prep => "себе";}

  Verb : Type = {s : ClForm => GenNum => Person => Str ;
                 asp : Aspect ; w: Voice} ;
```

Verb phrases are discontinuous: the parts of a verb phrase are (s) an inflected

verb, (s2) verb adverbials (not negation though), and (s3) complement. This discontinuity is needed in sentence formation to account for word order variations.

```
VerbPhrase : Type = Verb ** {s2: Str; s3 : Gender =>
         Number => Str ; negBefore: Bool} ;
```

This is one instance of Gazdar's **slash categories** [11], corresponding to his `S/NP`. We cannot have - nor would we want to have - a productive slash-category former. Perhaps a handful more will be needed.

Notice that the slash category has the same relation to sentences as transitive verbs have to verbs: it's like a **sentence taking a complement**.

```
 SlashNounPhrase = Clause ** Complement ;
 Clause = {s : Polarity => ClForm => Str} ;
```

This is the traditional `S -> NP VP` rule:

```
predVerbPhrase : NounPhrase -> VerbPhrase -> SlashNounPhrase =
 \Ya, tebyaNeVizhu -> { s = \\b,clf =>
 let
  {ya = Ya.s ! (mkPronForm Nom No NonPoss);
   khorosho = tebyaNeVizhu.s2;
   vizhu = tebyaNeVizhu.s!clf!(gNum (pgen2gen Ya.g) Ya.n)!Ya.p;
   tebya = tebyaNeVizhu.s3 ! (pgen2gen Ya.g) ! Ya.n
   }
  in
    ya ++  khorosho ++ vizhu ++ tebya;
  s2= "";
  c = Nom
} ;
```

Questions are either direct (*Ты счастлив?*) (*Are you happy?*) or indirect (*Потом он спросил счастлив ли ты*) (*Then he asked if you are happy*).

```
param QuestForm = DirQ | IndirQ ;
```

The order of sentence is needed already in `VP`.

```
oper
 getActVerbForm : ClForm -> Gender -> Number -> Person ->
  VerbForm = \clf,g,n, p -> case clf of
   {ClIndic Future _ => VFORM Act (VIND (gNum g n) (VFuture p));
    ClIndic Past _ => VFORM Act (VIND (gNum g n) VPast);
    ClIndic Present _ => VFORM Act (VIND (gNum g n) (VPresent p));
    ClCondit => VFORM Act (VSUB (gNum g n));
    ClInfinit => VFORM Act VINF ;
    ClImper => VFORM Act (VIMP n p)
 };
```

2.4 For `Adjective`

Adjective declination forms depend on Case, Animacy, Gender and Number: *большие дома - больших домов* (*big houses - big houses'*); and on Number: *большой дом - большие дома* (*a big house - big houses*); Animacy plays role only in the Accusative case (Masc Sg and Plural forms): Accusative Animate = Genitive, Accusaive Inanimate = Nominative *я люблю большие дома - я люблю больших мужчин* (*I love big houses - I love big men*). `AdvF` is a special adverbial form: *хороший - хорошо* (*good-well*):

```
param
  AdjForm = AF Case Animacy GenNum | AdvF;

oper
  Complement =  {s2 : Str ; c : Case} ;

  pgNum : PronGen -> Number -> GenNum = \g,n ->
    case n of
   {   Sg => case g of
                { PGen Fem => ASg Fem ;
                  PGen Masc => ASg Masc ;
                  PGen Neut => ASg Neut ;
                  _ => ASg Masc } ;
        Pl => APl
   } ;

oper numGNum : GenNum -> Number = \gn ->
   case gn of { APl => Pl ; _ => Sg } ;

oper genGNum : GenNum -> Gender = \gn ->
   case gn of { ASg Fem => Fem; ASg Masc => Masc; _ => Neut } ;

oper numAF: AdjForm -> Number = \af ->
   case af of { AdvF => Sg; AF _ _  gn => (numGNum gn) } ;

oper genAF: AdjForm -> Gender = \af ->
   case af of { AdvF => Neut; AF _ _  gn => (genGNum gn) } ;

oper caseAF: AdjForm -> Case = \af ->
   case af of { AdvF => Nom; AF c _ _ => c } ;
```

The Degree parameter should also be more complex, since most Russian adjectives have two comparative forms: attributive (syntactic (compound), declinable) - *более высокий* (corresponds to *more high*) and predicative (indeclinable)- *выше*

(*higher*) and more than one superlative forms: *самый высокий* (corresponds to *the most high*) - *высочайший/высший* (*the highest*). Even one more parameter independent of the degree can be added, since Russian adjectives in the positive degree also have two forms: long (attributive and predicative) - *высокий* (*high*) and short (predicative) - *высок*, although this parameter will not be exactly orthogonal to the degree parameter. Short form has no case declension, so in principle it can be considered as an additional case. Note: 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: *он - больной* (long, predicative) vs. *он - болен* (short, predicative).

Adjective phrases

An adjective phrase may contain a complement, e.g. *моложе Риты* (*younger than Rita*). Then it is used as postfix in modification, e.g. *человек, моложе Риты* (*a man younger than Rita*).

```
IsPostfixAdj = Bool ;
```

Simple adjectives are not postfix: Adjective type includes both non-degree adjective classes: possesive (*мамин* [*mother's*], *лисий* [*fox'es*]) and relative (*русский* [*Russian*]) adjectives.

```
Adjective : Type = {s : AdjForm => Str} ;
```

A special type of adjectives just having positive forms (for semantic reasons) is useful, e.g. *финский* (*Finnish*).

```
AdjPhrase = Adjective ** {p : IsPostfixAdj} ;

mkAdjPhrase : Adjective -> IsPostfixAdj  -> AdjPhrase =
 \novuj ,p -> novuj ** {p = p} ;
```

2.5 For `Numeral`

Parameters and operations for cardinal numbers:

```
param DForm = unit  | teen  | ten | hund ;
param Place = attr  | indep  ;
param Size  = nom | sgg | plg ;

oper mille : Size => Str = table {
   {nom} => "тысяча" ;
   {sgg} => "тысячи" ;
   _ => "тысяч"} ;
 oper gg : Str -> Gender => Str = \s -> table {_ => s} ;
```

2.6 Transformations between parameter types

Extracting `Number` value from `SubstForm` (see subsection 2.2) value:

```
  oper
     numSF: SubstForm -> Number = \sf -> case sf of
      {
        SF Sg _ => Sg ;
        _       => Pl
      } ;
```

Extracting `Case` value from `SubstForm` (see subsection 2.2) value:

```
     caseSF: SubstForm -> Case = \sf -> case sf of
      {
        SF _ Nom => Nom ;
        SF _ Gen => Gen ;
        SF _ Dat => Dat ;
        SF _ Inst => Inst ;
        SF _ Acc => Acc ;
        SF _ Prepos => Prepos
      } ;
}
```

3 Categories

3.1 Abstract API

The category system is central to the library in the sense that the other modules (`Adjective`, `Adverb`, `Noun`, `Verb` etc) communicate through it. This means that a e.g. a function using `NP`s in `Verb` need not know how `NP`s are constructed in `Noun`: it is enough that both `Verb` and `Noun` use the same type `NP`, which is given here in `Cat`.

Some categories are inherited from `Common`. The reason they are defined there is that they have the same implementation in all languages in the resource (typically, just a string). These categories are `AdA, AdN, AdV, Adv, Ant, CAdv, IAdv, PConj, Phr, Pol, SC, Tense, Text, Utt, Voc`.

Moreover, the list categories `ListAdv, ListAP, ListNP, ListS` are defined on `Conjunction` and only used locally there.

```
  abstract Cat = Common ** {

    cat
```

Sentences and clauses

Constructed in `Sentence`, and also in `Idiom`.

```
S ;     -- declarative sentence
        -- e.g. "she lived here"
```
”она жила здесь”
```
QS ;    -- question
        -- e.g. "where did she live"
```
”где она жила ”
```
RS ;    -- relative
        -- e.g. "in which she lived"
```
”в котором она жила”
```
Cl ;    -- declarative clause, with all tenses
        -- e.g. "she looks at this"
```
”она смотрит на это”
```
Slash ; -- clause missing NP (S/NP in GPSG)
        -- e.g. "she looks at"
```
”она смотрит на”
```
Imp ;   -- imperative
        -- e.g. "look at this"
```
”смотри на это”

Questions and interrogatives

Constructed in `Question`.

```
QCl ;   -- question clause, with all tenses
        -- e.g. "why does she walk"
```
”почему она идёт”
```
IP ;    -- interrogative pronoun
        --  e.g. "who"
```
”кто”
```
IComp ; -- interrogative complement of copula
        -- e.g. "where"
```
”где”
```
IDet ;  -- interrogative determiner
        -- e.g. "which"
```
”который”

Relative clauses and pronouns

Constructed in `Relative`.

```
RCl ;   -- relative clause, with all tenses
        -- e.g. "in which she lives"
```

”в котором она живёт”

```
RP ;    -- relative pronoun
        --  e.g. "in which"
```

”в котором”

Verb phrases

Constructed in `Verb`.

```
VP ;    -- verb phrase
        -- e.g. "is very warm"
```

”очень тёплый”

```
Comp ;  -- complement of copula, such as AP
        -- e.g. "very warm"
```

”очень тёплый”

Adjectival phrases

Constructed in `Adjective`.

```
AP ;    -- adjectival phrase
        -- e.g. "very warm"
```

”очень тёплый”

Nouns and noun phrases

Constructed in `Noun`. Many atomic noun phrases e.g. *everybody* are constructed in `Structural`. The determiner structure is

```
Predet (QuantSg | QuantPl Num) Ord
```

as defined in `Noun`.

```
CN ;    -- common noun (without determiner)
        -- e.g. "red house"
```

”красный дом”

```
NP ;    -- noun phrase (subject or object)
        -- e.g. "the red house"
```

"красный дом"

```
Pron ;  -- personal pronoun
        -- e.g. "she"
```

"она"

```
Det ;   -- determiner phrase
        -- e.g. "all the seven"
```

"все семь"

```
Predet; -- predeterminer (prefixed Quant)
        -- e.g. "all"
```

"все"

```
QuantSg;-- quantifier ('nucleus' of sing. Det)
        -- e.g. "every"
```

"каждый"

```
QuantPl;-- quantifier ('nucleus' of plur. Det)
        -- e.g. "many"
```

"много"

```
Quant ; -- quantifier with both sg and pl
        -- e.g. "this/these"
```

"этот/эти"

```
Num ;   -- cardinal number (used with QuantPl)
        -- e.g. "seven"
```

"семь"

```
Ord ;   -- ordinal number (used in Det)
        -- e.g. "seventh"
```

"седьмой"

Numerals

Constructed in `Numeral`.

```
Numeral; -- cardinal or ordinal, e.g. "five/fifth"
```

"пять/пятый"

Structural words

Constructed in `Structural`.

```
Conj ;  -- conjunction,          e.g. "and"
                                      "и"
DConj ; -- distributed conj.     e.g. "both - and"
                                      "как - так"
Subj ;  -- subjunction,          e.g. "if"
                                      "если"
Prep ;  -- preposition, or just case e.g. "in"
                                      "в"
```

Words of open classes

These are constructed in `Lexicon` and in additional lexicon modules.

```
V ;     -- one-place verb        e.g. "sleep"
                                      "спать"
V2 ;    -- two-place verb        e.g. "love"
                                      "любить"
V3 ;    -- three-place verb      e.g. "show"
                                      "показывать"
VV ;    -- verb-phrase-complement verb e.g. "want"
                                      "хотеть"
VS ;    -- sentence-complement verb    e.g. "claim"
                                      "утверждать"
VQ ;    -- question-complement verb    e.g. "ask"
                                      "спрашивать"
VA ;    -- adjective-complement verb   e.g. "look"
                                      "смотреть"
```

```
    V2A ;   -- verb with NP and AP complement e.g. "paint"
                                              "рисовать"
    A ;     -- one-place adjective          e.g. "warm"
                                              "тёплый"
    A2 ;    -- two-place adjective          e.g. "divisible"
                                              "делимый"
    N ;     -- common noun                  e.g. "house"
                                              "дом"
    N2 ;    -- relational noun              e.g. "son"
                                              "сын"
    N3 ;    -- three-place relational noun e.g. "connection"
                                              "связь"
    PN ;    -- proper name                  e.g. "Paris"
                                              "Париж"
}
```

3.2 Russian Implementation

Parameters are listed and explained in the Resource section 2 or belong to the language-independent `CommonX` module.

```
concrete CatRus of Cat = CommonX ** open ResRus, Prelude in {
```

The flag `optimize=all_subs` turns on the so called subexpression elimination that eliminates repetition of the same expressions, which can result in reducing the grammar size up to 90% of the original. For an example of a typical case where such optimization is useful, see technical report B subsection 2.1:

```
  flags optimize=all_subs ; coding=utf8 ;

  lincat
```

Phrase

```
    Utt, Voc = {s : Str} ;
```

Tensed/Untensed

```
S  =  {s : Str} ;
SC = {s : Str} ;
QS =  {s : QForm => Str} ;
-- QForm distinguish between direct or indirect
RS =  {s : GenNum => Case => Animacy => Str} ;
```

Many `RS` forms are due to Russian declension of *который* (*which*), which declines like an adjective.

Sentence

```
Cl    = {s : Polarity => ClForm => Str} ;
Slash = {s : Polarity => ClForm => Str; s2: Str; c: Case} ;
Imp   = {s : Polarity => Gender => Number => Str } ;
```

Polarity stands for either negative or positive statements. In imperative `Gender` and `Number` is needed for verb conjugation, while in `Slash` and `Clause` they are included in the `ClForm`-parameter.

Question

```
QCl = {s :Polarity => ClForm => QForm => Str};
IP  = {s : PronForm => Str ; n : Number ; p : Person ;
       g: PronGen ; anim : Animacy ;  pron: Bool} ;
IAdv, IComp = {s : Str} ;
IDet = Adjective ** {n: Number; g: PronGen; c: Case} ;
```

Determiners inflect like non-degree adjectives.

Relative

```
RCl = {s : Polarity => ClForm => GenNum => Case => Animacy
       => Str} ;
RP  = {s :   GenNum => Case => Animacy => Str} ;
```

Similarly to `RS`, many `RS` forms are due to Russian declension of *который* (*which*).

Verb

```
Comp, VP = {s : ClForm => GenNum => Person => Str ;
            asp : Aspect ; w: Voice ; s2 : Str ;
          s3 : Gender => Number => Str ; negBefore: Bool} ;
```

Verb phrases are discontinuous: the parts of a verb phrase are (`s`) an inflected verb, (`s2`) verb adverbials (not negation though), and (`s3`) complement. This discontinuity is needed in sentence formation to account for word order variations.

Adjective

```
AP = {s : AdjForm => Str; p : IsPostfixAdj} ;
```

An adjective phrase may contain a complement, e.g. *моложе Риты* (*younger than Rita*). Then it is used as postfix in modification, e.g. *человек, моложе Риты* (*a man, younger than Rita*).

Noun

```
CN = {s : Number => Case => Str; g : Gender; anim : Animacy} ;
NP = { s : PronForm => Str ; n : Number ; p : Person ;
       g: PronGen ; anim : Animacy ;  pron: Bool} ;
Pron = { s : PronForm => Str ; n : Number ; p : Person ;
         g: PronGen ;  pron: Bool} ;
```

`CN` basically corresponds to a noun lexicon entry `N`. `NP` is based on `Pron` type, so except for `Animacy` field they are the same.

Determiners (only determinative pronouns (or even indefinite numerals: *много* (*many*)) in Russian) are inflected according to the gender of nouns they determine. extra parameters (`Number` and `Case`) are added for the sake of the determinative pronoun *большинство* (*most*); `Gender` parameter is due to multiple determiners (`Numerals` in Russian) like *много*. The determined noun has the case parameter specific for the determiner:

```
QuantSg, QuantPl , Det = {s : AdjForm => Str; n: Number;
                          g: PronGen; c: Case} ;
Predet, Quant = {s : AdjForm => Str; g: PronGen; c: Case} ;
```

Adverb

```
Adv, AdV, AdA, AdS, AdN = {s : Str} ;
```

Numeral

```
Num, Numeral = {s : Case => Gender => Str} ;
```

Structural

The conjunction has an inherent number, which is used when conjoining noun phrases: *Иван и Маша поют* (*John and Mary sing*) vs. **Иван и Маша поёт* (**John and Mary sings*); in the case of *или*, the result is, however, plural if any of the disjuncts is.

```
Conj = {s : Str ; n : Number} ;
DConj = {s1,s2 : Str ; n : Number} ;
PConj = {s : Str} ;
CAdv = {s : Str} ;
Subj = {s : Str} ;
Prep = {s : Str ; c: Case } ;
```

Open lexical classes, e.g. Lexicon.

This is the verb type used in the lexicon:

```
V, VS, VV, VQ, VA = Verbum ;
  -- = {s : VerbForm => Str ; asp : Aspect } ;
V2, V2A = Verbum ** Complement ;
V3 = Verbum ** Complement** {s4 : Str; c2: Case} ;
VV = {s : VVForm => Str ; isAux : Bool} ;
```

`Aspect` and `Voice` parameters are present in every mood, so `Voice` is put before the mood parameter in verb form description the hierachy. Moreover `Aspect` is regarded as an inherent parameter of a verb entry. The primary reason for that is that one imperfective form can have several perfective forms: *ломать* - *сломать* - *поломать* (*to break*). Besides, the perfective form could be formed from imperfective by prefixation, but also by taking a completely different stem: *говорить-сказать* (*to say*). In the later case it is even natural to regard them as different verb entries. Another reason is that looking at the `Aspect` as an inherent verb parameter seem to be customary in other similar projects, see [34].

Similarly participles (`Present` and `Past`) and `Gerund` forms are better to be handled separately (not included in the current description), since they decline like non-degree adjectives and having them together with verb forms would be too much for a single lexicon entry. Such separation is, however, non-standard and does not present in the GF resource grammars for other languages.

```
    -- Ordinals decline like non-degree adjectives.
    Ord = {s : AdjForm => Str} ;
    A =  {s : Degree => AdjForm => Str} ;
    A2 = A ** Complement ;

    -- Substantives moreover have an inherent gender.
    N = {s : SubstForm => Str ; g : Gender ; anim : Animacy } ;
    N2 = {s : Number => Case => Str; g : Gender; anim : Animacy}
          ** Complement ;
    N3 = {s : Number => Case => Str; g : Gender; anim : Animacy}
          ** Complement ** {s3 : Str; c2: Case} ;
    PN = {s :  Case => Str ; g : Gender ; anim : Animacy} ;
}
```

4 Adjective

4.1 Abstract API

Language-independent functions (abstract syntax) for forming adjective phrases.

```
abstract Adjective = Cat ** {

  fun
```

The principal ways of forming an adjectival phrase are positive, comparative, relational, reflexive-relational, and elliptic-relational. (The superlative use is covered in `Noun.SuperlA`.)

```
    PositA  : A -> AP ;          -- warm

                                    тёплый

    ComparA : A -> NP -> AP ;    -- warmer than Spain

                                    теплее, чем Испания

    ComplA2 : A2 -> NP -> AP ;   -- divisible by 2

                                    делится на 2

    ReflA2  : A2 -> AP ;         -- divisible by itself

                                    делится на себя

    UseA2   : A2 -> A ;          -- divisible
```

делится

Sentence and question complements defined for all adjectival phrases, although the semantics is only clear for some adjective.

```
    SentAP  : AP -> SC -> AP ;  -- great that she won,
                                -- uncertain if she did
```

здорово, что она выйграла,
если это правда

An adjectival phrase can be modified by an **adadjective**, such as *very*.

```
    AdAP    : AdA -> AP -> AP ; -- very uncertain
```

совсем не уверен

The formation of adverbs from adjective (e.g. *quickly*) is covered by `Adverb`.

```
}
```

4.2 Russian Implementation

Russian implementation of `Adjective` API (concrete syntax for Russian).

```
concrete AdjectiveRus of Adjective = CatRus ** open ResRus,
 Prelude in {
  flags  coding=utf8 ;

  lin
```

`False`-value of `p`-parameter indicates a prefix adjective:

```
    PositA  a = { s = a.s ! Posit; p = False} ;
```

Comparative forms are used with an object of comparison, as adjectival phrases (*больше тебя*) [*more than you*]:

```
    ComparA bolshoj tu =
      {s = \\af => bolshoj.s ! Compar ! af ++
            tu.s ! (mkPronForm Gen Yes NonPoss) ;
       p = True
      } ;
```

`SuperlA` belongs to determiner syntax in `Noun`.
`True`-value of `p`-parameter indicates a postfix adjective. Agreement: `vlublen.s2`-preposition requires `vlublen.c`-case of `tu.s`-object:

```
    ComplA2 vlublen tu =
      {s = \\af => vlublen.s ! Posit ! af ++ vlublen.s2 ++
            tu.s ! (mkPronForm vlublen.c No NonPoss) ;
       p = True
      } ;

    ReflA2 vlublen =

      {s = \\af => vlublen.s !Posit!  af ++ vlublen.s2 ++ "себя";

       p = True
      } ;

    SentAP vlublen sent =
      {s = \\af => vlublen.s ! af ++ [", "] ++ sent.s;
        p = True
      } ;

    AdAP ada ap =
      {s = \\af => ada.s ++ ap.s ! af ;
        p = True
      } ;

    UseA2 a = a ;
}
```

5 Adverb

5.1 Abstract API

Language-independent functions (abstract syntax) for forming adverbs.

```
  abstract Adverb = Cat ** {
    fun
```

The two main ways of forming adverbs are from adjectives and by prepositions from noun phrases.

```
      PositAdvAdj : A -> Adv ;                  -- quickly

                                                   быстро

      PrepNP      : Prep -> NP -> Adv ;         -- in the house
```

в доме

Comparative adverbs have a noun phrase or a sentence as object of comparison.

```
    ComparAdvAdj  : CAdv -> A -> NP -> Adv ; -- more quickly
                                             -- than John
```

быстрее,
чем Иван

```
    ComparAdvAdjS : CAdv -> A -> S -> Adv ;  -- more quickly
                                             -- than he runs
```

быстрее,
чем он бежит

Adverbs can be modified by 'adadjectives', just like adjectives.

```
    AdAdv  : AdA -> Adv -> Adv ;             -- very quickly
```

очень быстро

Subordinate clauses can function as adverbs.

```
    SubjS : Subj -> S -> Adv ;               -- when he arrives
```

когда он придёт

```
    AdvSC : SC -> Adv ;                      -- that he arrives
```

что он придёт

Comparison adverbs also work as numeral adverbs.

```
    AdnCAdv : CAdv -> AdN ;                  -- more (than five)
```

больше

```
}
```

5.2 Russian Implementation

Russian implementation of `Adverb` API (concrete syntax for Russian).

```
concrete AdverbRus of Adverb = CatRus ** open ResRus, Prelude in {
  flags  coding=utf8 ;

  lin
```

In the following three functions a positive (`Posit`) adverb form (`AdvF`) of `a.s`-adjective is used:

```
    PositAdvAdj a = {s = a.s !Posit! AdvF} ;

    ComparAdvAdj cadv a np =

      {s = cadv.s ++ a.s !  Posit !  AdvF ++ "чем" ++

          np.s ! PF Nom No NonPoss}
      } ;

    ComparAdvAdjS cadv a s =

      {s = cadv.s ++ a.s !  Posit !  AdvF ++ "чем" ++ s.s

      } ;
```

Agreement: `na`-preposition requires `na.c`-case of `stol.s`-object:

```
    PrepNP na stol = ss (na.s ++ stol.s ! PF na.c Yes NonPoss) ;

    AdAdv = cc2 ; -- concatenation of two strings

    SubjS = cc2 ;

    AdvSC s = s ;

    AdnCAdv cadv = {s = cadv.s ++ "чем"} ;
}
```

6 Conjunction

6.1 Abstract API

Coordination is defined for many different categories; here is a sample. The rules apply to **lists** of two or more elements, and define two general patterns: - ordinary conjunction: X,...X and X - distributed conjunction: both X,...,X and X

Note. This module uses right-recursive lists. If backward compatibility with API 0.9 is needed, use `SeqConjunction`.

```
  abstract Conjunction = Cat ** {
```

Rules

```
fun
  ConjS    : Conj -> [S] -> S ;     -- "John walks and Mary runs"
```

"Иван идет и Маша бежит"

```
  ConjAP   : Conj -> [AP] -> AP ;   -- "even and prime"
```
”чётный и простой”
```
  ConjNP   : Conj -> [NP] -> NP ;   -- "John or Mary"
```
”Иван и Маша”
```
  ConjAdv  : Conj -> [Adv] -> Adv ; -- "quickly or slowly"
```
”быстро или медленно”
```
  DConjS   : DConj -> [S] -> S ;    -- "either John walks or Mary runs"
```
”либо Иван идет,
либо Маша бежит”
```
  DConjAP  : DConj -> [AP] -> AP ;  -- "both even and prime"
```
”и чётный, и простой”
```
  DConjNP  : DConj -> [NP] -> NP ;  -- "either John or Mary"
```
”либо Иван, либо Маша”
```
  DConjAdv : DConj -> [Adv] -> Adv; -- "both badly and slowly"
```
”и плохо, и медленно”

Categories

These categories are only used in this module.

```
    cat
      [S]{2} ;
      [Adv]{2} ;
      [NP]{2} ;
      [AP]{2} ;
```

List constructors

The list constructors are derived from the list notation and therefore not given explicitly. But here are their type signatures:

```
    --  BaseC : C -> C   -> [C] ;  -- for C = S, AP, NP, Adv
    --  ConsC : C -> [C] -> [C] ;
  }
```

6.2 Russian Implementation

Most of the functions have generic implementations that use dependent types and, therefore, the same for all languages and all categories.

```
concrete ConjunctionRus of Conjunction =
  CatRus ** open ResRus, Coordination, Prelude in {

  flags optimize=all_subs ;  coding=utf8 ;

  lin

    ConjS = conjunctSS ;
    DConjS = conjunctDistrSS ;

    ConjAdv = conjunctSS ;
    DConjAdv = conjunctDistrSS ;

    ConjNP c xs =
      conjunctTable PronForm c xs ** {n = conjNumber c.n xs.n ;
      anim = xs.anim ;
       p = xs.p; g = xs.g ; pron = xs.pron} ;

    DConjNP  c xs =
    conjunctDistrTable PronForm c xs ** {n = conjNumber c.n xs.n ;
      p = xs.p ; pron = xs.pron ; anim = xs.anim ;
       g = xs.g } ;

    ConjAP c xs =    conjunctTable AdjForm c xs ** {p = xs.p} ;

    DConjAP c xs = conjunctDistrTable AdjForm c xs ** {p = xs.p} ;
```

These fun's are generated from the list cat's.

```
    BaseS = twoSS ;
    ConsS = consrSS comma ;
    BaseAdv = twoSS ;
    ConsAdv = consrSS comma ;

    ConsNP  x xs =
    consTable PronForm comma xs x **
       {n = conjNumber xs.n x.n ; g = conjPGender x.g xs.g ;
          anim = conjAnim x.anim xs.anim ;
          p = conjPerson xs.p x.p; pron = conjPron xs.pron x.pron} ;
```

```
    ConsAP x xs = consTable AdjForm comma xs x ** {p = andB xs.p x.p} ;

    BaseAP x y = twoTable AdjForm x y ** {p = andB x.p y.p} ;

    BaseNP x y = twoTable PronForm x y ** {n = conjNumber x.n y.n ;
       g = conjPGender x.g y.g ; p = conjPerson x.p y.p ;
       pron = conjPron x.pron y.pron ; anim = conjAnim x.anim y.anim } ;

  lincat
    [S] = {s1,s2 : Str} ;
    [Adv] = {s1,s2 : Str} ;
```

The structure is the same as for sentences. The result is either always plural or plural if any of the components is, depending on the conjunction:

```
    [NP] = { s1,s2 : PronForm => Str ; g: PronGen ;
     anim : Animacy ; n : Number ; p : Person ;  pron : Bool } ;
```

The structure is the same as for sentences. The result is a prefix adjective if and only if all elements are prefix:

```
    [AP] =  {s1,s2 : AdjForm => Str ; p : Bool} ;
```

We have to define a calculus of numbers of persons. For numbers, it is like the conjunction with `Pl` corresponding to `False`.

```
oper
  conjNumber : Number -> Number -> Number = \m,n -> case <m,n> of {
      <Sg,Sg> => Sg ;
      _ => Pl
      } ;
```

For persons, we let the latter argument win (*?либо ты, либо я пойду* [*either you or I will go*], but *?либо ты, либо я пойдёшь* (*either you or I will go*)). This is not quite clear.

```
  conjPerson : Person -> Person -> Person = \_,p -> p ;
```

For `Pron`, we let the latter argument win – **Маша или моя мама* (Nominative case) but – **моей или Машина мама* (Genitive case) both corresponds to *Masha's or my mother*, which is actually not exactly correct, since different cases should be used – *Машина или моя мама*.

```
  conjPron : Bool -> Bool -> Bool = \_,p -> p ;
```

For gender in a similar manner as for person: Needed for adjective predicates like: *Маша или Оля – красивая* (*Masha or Olya is beautiful*), *Антон или Олег – красивый* (*Anton or Oleg is beautiful*), **Маша или Олег – красивый* (*Masha or Oleg is beautiful*). The later is not totally correct, but there is no correct way to say that.

```
 conjGender : Gender -> Gender -> Gender = \_,m -> m ;
 conjPGender : PronGen -> PronGen -> PronGen = \_,m -> m ;
 conjAnim : Animacy -> Animacy -> Animacy = \_,m -> m ;
}
```

7 Idiom

7.1 Abstract API

This module defines constructions that are formed in fixed ways, often different even in closely related languages.

```
  abstract Idiom = Cat ** {

    fun
```

A separate example for Russian, since *it rains* can not be translated into an impersonal construction (*дождь идёт*):

```
      ImpersCl  : VP -> Cl ;    -- it rains
```

светает (dawn is breaking)

No generic clauses in Russian. The function is kept for the sake of compatibility with the language-independent API. Infinitive is the closest form, see also subsection 7.2 for more comments:

```
      GenericCl : VP -> Cl ;    -- one sleeps
```

спишь

```
      ExistNP   : NP -> Cl ;    -- there is a house
```

есть дом

```
      ExistIP   : IP -> QCl ;   -- which houses are there
```

какие есть дома

```
      ProgrVP   : VP -> VP ;    -- be sleeping
```

спит

```
    ImpPl1    : VP -> Utt ;   -- let's go
                                  давайте пойдём
    CleftNP   : NP  -> RS -> Cl ; -- it is you who did it
                                  это ты, кто сделал это
    CleftAdv  : Adv -> S  -> Cl ; -- it is yesterday she arrived
                                  это вчера, она вернулась
}
```

7.2 Russian Implementation

Russian implementation of `Idiom` API (concrete syntax for Russian). Every function has a positive (`Pos`) and a negative (`Neg`) version that normally differ by negation particle *не* (`ne`).

```
concrete IdiomRus of Idiom = CatRus ** open Prelude,
 ResRus, MorphoRus in {

 flags optimize=all_subs ;  coding=utf8 ;

 lin
```

The verb *существовать* (`suchestvovat.s`) in Russian corresponds to the verb *exist* in English. The verb is conjugated according to object's gender (`bar.g`, `Kto.g`), number (`bar.n, Kto.n`) and person (`P3`):

```
  ExistNP = \bar -> {s =\\b,clf =>
   let ne = case b of {
      Pos =>[];

      Neg =>"не"}

    in
   ne ++ verbSuchestvovat.s ! (getActVerbForm clf }
    (pgen2gen bar.g) bar.n P3) ++ bar.s ! PF Nom No NonPoss
  } ;

  ExistIP Kto = {s =  \\b,clf,_ =>
    let {
      kto = Kto.s ! (PF Nom No NonPoss);

       ne = case b of {Pos =>[]; Neg =>"не"}
```

```
    in
   kto ++ ne ++ verbSuchestvovat.s!
          (getActVerbForm clf (pgen2gen Kto.g) Kto.n P3)
} ;
```

Impersonal subject is omitted. The third person (P3) singular (ASg) verb form is used:

```
ImpersCl vp = {s= \\ b, clf => let ne = case b of

  {Pos =>[]; Neg => "не"}

   in
  ne ++ vp.s! clf! (ASg Neut) ! P3  };
```

No direct correspondence in Russian. Usually expressed by 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 transferred into adverb *можно* (*it is possible*) in Russian The closest subject is *ты* (*you*), which is omitted in the final sentence: *если очень захочешь, можешь в космос улететь*:

```
GenericCl vp = {s= \\ b, clf =>

  let ne= case b of {Pos =>[]; Neg =>"не"}

   in
  ne ++ vp.s! clf! (ASg Masc) ! P2 };

ProgrVP vp = vp ;
```

The English phrase *let us* corresponds to *давайте* in Russian:

```
ImpPl1 vp = {s = "давайте"++vp.s!(ClIndic Future Simul)!APl!P1};
```

The English phrase *it is* corresponds to *это* in Russian:

```
CleftAdv adv sen = {s= \\ b, clf =>
  let ne = case b of {
    Pos =>[];

    Neg =>"не"}

  in

 "это" ++ ne ++ adv.s ++ [", "]++ sen.s };

CleftNP np rs = {s= \\ b, clf =>
  let
```

```
      ne = case b of {Pos =>[]; Neg =>"не"};
      gn = case np.n of {Pl => APl; _=> ASg (pgen2gen np.g)}
    in
  "это" ++ ne ++ np.s!(PF Nom No NonPoss) ++ rs.s!gn!Nom!Animate};
}
```

8 Noun

8.1 Abstract API

Language-independent functions (abstract syntax) for forming noun phrases.

```
abstract Noun = Cat ** {
```

Noun phrases

The three main types of noun phrases are - common nouns with determiners - proper names - pronouns

```
fun
  DetCN   : Det -> CN -> NP ;   -- the man
                                   человек
  UsePN   : PN -> NP ;          -- John
                                   Иван
  UsePron : Pron -> NP ;        -- he
                                   он
```

Pronouns are defined in the module `Structural`. A noun phrase already formed can be modified by a `Predet`erminer.

```
  PredetNP : Predet -> NP -> NP; -- only the man
                                    только человек
```

A noun phrase can also be postmodified by the past participle of a verb or by an adverb.

```
  PPartNP : NP -> V2  -> NP ;    -- the number squared
                                    число в квадрате
  AdvNP   : NP -> Adv -> NP ;    -- Paris at midnight
                                    Париж ночью
```

Determiners

The determiner has a fine-grained structure, in which a 'nucleus' quantifier and two optional parts can be discerned. The cardinal numeral is only available for plural determiners. (This is modified from CLE by further dividing their `Num` into cardinal and ordinal.)

```
DetSg : QuantSg ->        Ord -> Det ;  -- this best man

                                           этот свидетель

DetPl : QuantPl -> Num -> Ord -> Det ;  -- these five best men

                                           эти пять свидетелей
```

Quantifiers that have both forms can be used in both ways.

```
SgQuant : Quant -> QuantSg ;            -- this

                                           этот

PlQuant : Quant -> QuantPl ;            -- these

                                           эти
```

Pronouns have possessive forms. Genitives of other kinds of noun phrases are not given here, since they are not possible in e.g. Romance languages.

```
PossPron : Pron -> Quant ;              -- my (house)

                                           мой (дом)
```

All parts of the determiner can be empty, except `Quant`, which is the *kernel* of a determiner.

```
NoNum  : Num ;

NoOrd  : Ord ;
```

`Num` consists of either digits or numeral words.

```
NumInt     : Int -> Num ;     -- 51

NumNumeral : Numeral -> Num ; -- fifty-one

                                  пятьдесят один
```

The construction of numerals is defined in `Numeral`. `Num` can be modified by certain adverbs.

```
AdNum : AdN -> Num -> Num ;   -- almost 51
```

почти пятьдесят один

`Ord` consists of either digits or numeral words.

```
OrdInt     : Int -> Ord ;     -- 51st
```

51-й

```
OrdNumeral : Numeral -> Ord ; -- fifty-first
```

пятьдесят первый

Superlative forms of adjectives behave syntactically in the same way as ordinals.

```
OrdSuperl : A -> Ord ;        -- largest
```

самый большой

Definite and indefinite constructions are sometimes realized as neatly distinct words (Spanish *un, unos ; el, los*) but also without any particular word (Finnish; Swedish definites).

```
DefArt   : Quant ;            -- the (house), the (houses)
```

дом, дома

```
IndefArt : Quant ;            -- a (house), (houses)
```

дом, дома

Nouns can be used without an article as mass nouns. The resource does not distinguish mass nouns from other common nouns, which can result in semantically odd expressions.

```
MassDet  : QuantSg ;          -- (beer)
```

пиво

Other determiners are defined in `Structural`.

Common nouns

Simple nouns can be used as nouns outright.

```
UseN : N -> CN ;              -- house
```

дом

Relational nouns take one or two arguments.

```
ComplN2 : N2 -> NP -> CN ;    -- son of the king
```

сын короля

```
    ComplN3 : N3 -> NP -> N2 ;    -- flight from Moscow (to Paris)
```

рейс из Москвы в Париж

Relational nouns can also be used without their arguments. The semantics is typically derivative of the relational meaning.

```
    UseN2   : N2 -> CN ;          -- son
```

сын

```
    UseN3   : N3 -> CN ;          -- flight
```

рейс

Nouns can be modified by adjectives, relative clauses, and adverbs (the last rule will give rise to many 'PP attachement' ambiguities when used in connection with verb phrases).

```
    AdjCN   : AP -> CN  -> CN ;   -- big house
```

большой дом

```
    RelCN   : CN -> RS  -> CN ;   -- house that John owns
```

дом, которым владеет Иван

```
    AdvCN   : CN -> Adv -> CN ;   -- house on the hill
```

дом на холме

Nouns can also be modified by embedded sentences and questions. For some nouns this makes little sense, but we leave this for applications to decide. Sentential complements are defined in `Verb`.

```
    SentCN  : CN -> SC  -> CN ;
              -- fact that John smokes, question if he does
```

факт, что Иван курит, если это так

Apposition

This is certainly overgenerating.

```
    ApposCN : CN -> NP -> CN ;    -- number x, numbers x and y
```

число x, числа x и y

```
}
```

8.2 Russian Implementation

Russian implementation of `Noun` API (concrete syntax for Russian).

```
concrete NounRus of Noun = CatRus ** open ResRus, Prelude,
  MorphoRus in {

  flags optimize=all_subs ;

  lin
```

The difference between Nominative (`Nom`) and other branches of determiner's (`kazhduj.c`) case is the choice of the case of the common noun (`okhotnik.s`). Number (`n`) parameter is inherited from the determiner. So is gender (`g`) parameter, except the case where the determiner has no gender-value, then the gender is taken from the common noun (`okhotnik`). Animacy (`anim`) is inherited from the common noun (`okhotnik`). `False`-value of `pron` parameter indicates that the result is not a pronoun (which is important if we need possessive forms). The result is in the third person (`p`):

```
    DetCN kazhduj okhotnik = {
        s = \\c => case kazhduj.c of {
         Nom =>
          kazhduj.s ! AF (extCase c) okhotnik.anim (gNum okhotnik.g
           kazhduj.n) ++ okhotnik.s ! kazhduj.n ! (extCase c) ;
         _ =>
          kazhduj.s ! AF (extCase c) okhotnik.anim (gNum okhotnik.g
           kazhduj.n) ++ okhotnik.s ! kazhduj.n ! kazhduj.c };
        n = kazhduj.n ;
        p = P3 ;
        pron = False;
        g = case kazhduj.g of {
           PNoGen => (PGen okhotnik.g); -- no gender-value case
           _ => kazhduj.g };
        anim = okhotnik.anim
      } ;
```

Animacy (`anim`) and gender (`g`) parameters are inherited from proper noun (`masha`). `False`-value of `pron` parameter indicates that the result is not a pronoun. The result is in the third person (`p`), singular number (`n`):

```
    UsePN masha = {
        s = \\pf => masha.s ! (extCase pf) ;
        p = P3;
```

```
        g = PGen masha.g ;
        anim = masha.anim ;
        n = Sg;
        pron = False
    } ;
```

Animacy (`anim`) is inanimate by default:

```
    UsePron p = p ** {anim = Inanimate};
```

All parameters are inherited from noun phrase argument (`np`). Predeterminer (`pred.s`) agrees with the noun phrase in gender (`np.g`), number (`np.n`) and animacy (`np.anim`):

```
    PredetNP pred np = {
        s = \\pf => pred.s! (AF (extCase pf) np.anim
         (gNum (pgen2gen np.g) np.n))++ np.s ! pf ;
        n = np.n;
        p = np.p;
        g = np.g;
        anim = np.anim;
        pron = np.pron
        } ;
```

All parameters are inherited from quantifier:

```
    DetSg quant ord = {
        s = \\af => quant.s!af ++ ord.s!af ;
        n = quant.n;
        g = quant.g;
        c = quant.c
        } ;
```

The resulting number is plural (`Pl`). Other parameters are inherited from quantifier:

```
    DetPl quant num ord = {
        s =  \\af => quant.s !af ++ num.s! (caseAF af) !
         (genAF af)  ++ ord.s!af ;
        n = Pl;
        g = quant.g;
        c = quant.c
        } ;
```

The resulting number is singular (`Sg`). Other parameters are inherited from quantifier:

```
SgQuant quant = {s = quant.s; c=quant.c; g=quant.g; n= Sg} ;
```

The resulting number is plural (`Pl`). Other parameters are inherited from quantifier:

```
PlQuant quant = {s = quant.s ; c=quant.c; g=quant.g; n= Pl} ;
```

Case parameter (`c`) is nominative (`Nom`), gender parameter (`g`) is unspecified (`PNoGen`) in the result:

```
PossPron p = {s = \\af => p.s ! mkPronForm (caseAF af) No
 (Poss (gNum (genAF af) (numAF af) )); c=Nom; g = PNoGen} ;

NoNum = {s = \\_,_ => []} ; -- cardinal numeral
NoOrd = {s = \\_ => []} ;   -- adjective
```

It is unclear how to tell apart the numbers from their string representation, so just leave a decimal representation, without case-suffixes:

```
NumInt i =  {s = table { _ => table {_ => i.s } } } ;
```

Ordinals inflect differently depending on the last digit. The functions `OrdNumeral` and `OrdInt` are not yet implemented for Russian.

```
NumNumeral n = n ;

AdNum adn num = {s = \\c,n => adn.s ++ num.s!c!n} ;

OrdSuperl a = {s = a.s!Posit};
```

No articles in Russian:

```
DefArt = {s = \\_=>[] ; c=Nom; g = PNoGen };
IndefArt = { s = \\_=>[] ; c=Nom; g = PNoGen };
MassDet = {s = \\_=>[] ; c=Nom; g = PNoGen; n = Sg} ;
```

All parameters are inherited from noun argument (`sb`):

```
UseN  sb =
  {s = \\n,c => sb.s ! SF n c ;
   g = sb.g ;
   anim = sb.anim
  } ;
```

It is possible to use a function word as a common noun; the semantics is often existential or indexical:

```
UseN2 x = x ;
UseN3 x = x ;
```

The application of a function gives, in the first place, a common noun: *ключ от дома* (*the key from the house*). From this, other rules of the resource grammar give noun phrases, such as *ключи от дома* (*the keys from the house*), *ключи от дома и от машины* (*the keys from the house and the car*), and *ключ от дома и от машины* (*the key from the house and the car*) (the latter two corresponding to distributive and collective functions, respectively). Semantics will eventually tell when each of the readings is meaningful:

```
ComplN2 mama ivan =
  {s = \\n, cas =>  case ivan.pron of
   { True => ivan.s ! (mkPronForm cas No (Poss (gNum mama.g n)))
          ++ mama.s ! n ! cas;
     False => mama.s ! n ! cas ++ mama.s2 ++
       ivan.s ! (mkPronForm mama.c Yes (Poss (gNum mama.g n)))
   };
   g = mama.g ;
   anim = mama.anim
 } ;
```

Two-place functions add one argument place. There application starts by filling the first place:

```
ComplN3 poezd paris =
  {s  = \\n,c => poezd.s ! n ! c ++ poezd.s2 ++ paris.s !
   (PF poezd.c Yes NonPoss) ;
   g  = poezd.g ; anim = poezd.anim;
   s2 = poezd.s3; c = poezd.c2
  } ;
```

The two main functions of adjective are in predication (*Иван – молод*) [*Ivan is young*] and in modification (*молодой человек*) [*a young man*]. Predication will be defined later, in the chapter on verbs:

```
AdjCN khoroshij novayaMashina =
  {s = \\n, c =>
       khoroshij.s ! AF c novayaMashina.anim
        (gNum novayaMashina.g n) ++ novayaMashina.s ! n ! c ;
       g = novayaMashina.g ;
       anim = novayaMashina.anim
   } ;
```

This is a source of the *I saw a man with a telescope*-ambiguity, and may produce strange things, like *машины всегда* (*cars always*). Semantics will have to make finer distinctions among adverbials:

```
AdvCN chelovek uTelevizora =
  {s = \\n,c => chelovek.s ! n ! c ++ uTelevizora.s ;
   g = chelovek.g ;
   anim = chelovek.anim
  } ;
```

Constructions like *идея, что два чётное* (*the idea that two is even*) are formed at the first place as common nouns, so that one can also have *предположение, что...* (*the suggestion that...*):

```
SentCN idea x = {
   s = \\n,c => idea.s ! n ! c ++ x.s ;
   g = idea.g;
   anim = idea.anim
  } ;
```

The relative clause (`x.s`) agrees with the common noun (`idea`) in gender (`idea.g`) and animacy (`idea.anim`). All parameters are inherited from the common noun argument (`idea`):

```
RelCN idea x = {
   s = \\n,c => idea.s ! n ! c ++
     x.s ! (gNum idea.g n) ! c ! idea.anim ;
   g = idea.g;
   anim = idea.anim
  } ;
```

All parameters are inherited from common noun argument (`cn`):

```
ApposCN cn s =
  {s = \\n,c => cn.s ! n ! c ++ s.s! PF c No NonPoss ;
   g = cn.g ;
   anim = cn.anim
  } ;
}
```

9 Numeral

9.1 Abstract API

Language-independent functions (abstract syntax) for forming numerals from 1 to 999999. The implementations are adapted from the numerals library, which

defines numerals for 88 languages. The resource grammar implementations add to this inflection (if needed) and ordinal numbers.

Note 1. Number 1 as defined in the category `Numeral` here should not be used in the formation of noun phrases, and should therefore be removed. Instead, one should use `Structural.one_Quant`. This makes the grammar simpler because we can assume that numbers form plural noun phrases.

Note 2. The implementations introduce spaces between parts of a numeral, which is often incorrect – more work on (un)lexing is needed to solve this problem.

```
abstract Numeral = Cat ** {

cat
  Digit ;       -- 2..9
  Sub10 ;       -- 1..9
  Sub100 ;      -- 1..99
  Sub1000 ;     -- 1..999
  Sub1000000 ;  -- 1..999999

fun
  num : Sub1000000 -> Numeral ;

  n2, n3, n4, n5, n6, n7, n8, n9 : Digit ;

  pot01 : Sub10 ;                                  -- 1
  pot0 : Digit -> Sub10 ;                          -- d * 1
  pot110 : Sub100 ;                                -- 10
  pot111 : Sub100 ;                                -- 11
  pot1to19 : Digit -> Sub100 ;                     -- 10 + d
  pot0as1 : Sub10 -> Sub100 ;               -- coercion of 1..9
  pot1 : Digit -> Sub100 ;                         -- d * 10
  pot1plus : Digit -> Sub10 -> Sub100 ;            -- d * 10 + n
  pot1as2 : Sub100 -> Sub1000 ;             -- coercion of 1..99
  pot2 : Sub10 -> Sub1000 ;                        -- m * 100
  pot2plus : Sub10 -> Sub100 -> Sub1000 ;          -- m * 100 + n
  pot2as3 : Sub1000 -> Sub1000000 ;         -- coercion of 1..999
  pot3 : Sub1000 -> Sub1000000 ;                   -- m * 1000
  pot3plus : Sub1000 -> Sub1000 -> Sub1000000 ; -- m * 1000 + n
}
```

9.2 Russian Implementation

Russian implementation of `Numeral` API (concrete syntax for Russian)[6]. Cardinal numerals only. Cases are not included in the present description.

```
concrete NumeralRus of Numeral = CatRus ** open ResRus in {

flags  coding=utf8 ;

lincat
  Digit = {s : DForm => Gender => Str ; size : Size} ;
  Sub10 = {s : Place => DForm => Gender => Str ; size : Size} ;
  Sub100 = {s : Place => Gender => Str ; size : Size} ;
  Sub1000 = {s : Place => Gender => Str ; size : Size} ;
  Sub1000000 = {s : Gender => Str} ;

lin

  num x = {s = table{ _ => x.s }};

  n2 = {s = table {
            {unit} => table {

                {Fem} => "две" ;
                _ => "два" };
            {teen} => gg "двенадцать" ;
            {ten} => gg "двадцать" ;
            {hund} => gg "двести" };

        size = sgg} ;

  n3 = {s = table {

            {unit} => gg "три" ;
            {teen} => gg "тринадцать" ;
            {ten} => gg "тридцать" ;
            {hund} => gg "триста" };

        size = sgg} ;

  n4 = {s = table {
```

[6]Implemented by Arto Mustajoki and Aarne Ranta.

```
          {unit} => gg "четыре" ;
          {teen} => gg "четырнадцать" ;
          {ten} => gg "сорок" ;
          {hund} => gg "четыреста" };

      size = sgg} ;

  n5 = {s = table {

          {unit} => gg "пять" ;
          {teen} => gg "пятнадцать" ;
          {ten} => gg "пятьдесят" ;
          {hund} => gg "пятьсот" };

      size = plg} ;

  n6 = {s = table {

          {unit} => gg "шесть" ;
          {teen} => gg "шестнадцать" ;
          {ten} => gg "шестьдесят" ;
          {hund} => gg "шестьсот" };

      size = plg} ;

  n7 = {s = table {

          {unit} => gg "семь" ;
          {teen} => gg "семнадцать" ;
          {ten} => gg "семьдесят" ;
          {hund} => gg "семьсот" };

      size = plg} ;

  n8 = {s = table {

          {unit} => gg "восемь" ;
          {teen} => gg "восемнадцать" ;
          {ten} => gg "восемьдесят" ;
          {hund} => gg "восемьсот" };

      size = plg} ;

  n9 = {s = table {
```

```
           {unit} => gg "девять" ;
           {teen} => gg "девятнадцать" ;
           {ten} => gg "девяносто" ;
           {hund} => gg "девятьсот" };

       size = plg} ;

  pot01 = {s = table {
              {attr} => table {

                 {hund} => gg "сто" ;

                 _ => gg []} ;
              _ => table {

                 {hund} => gg "сто" ;

                 _ => table {

                    {Masc} => "один" ;
                    {Fem} => "одна" ;
                     _ => "одно"}

              }
            } ;
          size = nom} ;

  pot0 d =
    {s = table {_ => d.s} ; size = d.size} ;
  pot110  =

    {s = table {_ => gg "десять" } ; size = plg} ;

  pot111  =

    {s = table {_ => gg "одиннадцать" } ; size = plg} ; -- 11

  pot1to19 d =
    {s = table {_ => d.s ! teen} ; size = plg} ;
  pot0as1 n =
    {s = table {p => n.s ! p ! unit} ; size = n.size} ;
  pot1 d =
    {s = table {_ => d.s ! ten} ; size = plg} ;
  pot1plus d e =
    {s = table {_ =>
           table {g => d.s ! ten ! g ++ e.s ! indep ! unit ! g}};
     size = e.size} ;
```

```
  pot1as2 n =
    {s = n.s ; size = n.size} ;
  pot2 d =
    {s = table {p => d.s ! p ! hund} ; size = plg} ;
  pot2plus d e =
    {s = table {
       p => table {g => d.s ! p ! hund ! g ++ e.s ! indep ! g}};
     size = e.size} ;
  pot2as3 n =
    {s = n.s ! indep} ;
  pot3 n =
    {s = gg (n.s ! attr ! Fem ++ mille ! n.size)} ;
  pot3plus n m =
    {s = table {
       g => n.s ! attr ! Fem ++ mille ! n.size ++ m.s ! indep ! g}
    } ;
  }
```

10 Phrase

10.1 Abstract API

Language-independent functions (abstract syntax) for forming phrases.

```
abstract Phrase = Cat ** {
```

When a phrase is built from an utterance it can be prefixed with a phrasal conjunction (such as *but*, *therefore*) and suffixing with a vocative (typically a noun phrase).

```
fun
  PhrUtt   : PConj -> Utt -> Voc -> Phr ; -- But go home my friend.
```

Но иди домой, мой друг.

Utterances are formed from sentences, questions, and imperatives.

```
  UttS     : S -> Utt ;                  -- John walks
```

Иван идёт

```
  UttQS    : QS -> Utt ;                 -- is it good
```

это хорошо

```
  UttImpSg : Pol -> Imp -> Utt;          -- (don't) help yourself
```

```
                                          (не) входи

                                          ((don't) come in)
  UttImpPl : Pol -> Imp -> Utt;           -- (don't) help yourselves

                                          (не) входите

                                          ((don't) come in)
```

There are also 'one-word utterances'. A typical use of them is as answers to questions. **Note**. This list is incomplete. More categories could be covered. Moreover, in many languages e.g. noun phrases in different cases can be used.

```
    UttIP   : IP   -> Utt ;               -- who

                                             кто

    UttIAdv : IAdv -> Utt ;               -- why

                                             почему

    UttNP   : NP   -> Utt ;               -- this man

                                             этот человек

    UttAdv  : Adv  -> Utt ;               -- here

                                             здесь

    UttVP   : VP   -> Utt ;               -- to sleep

                                             спать
```

The phrasal conjunction is optional. A sentence conjunction can also used to prefix an utterance.

```
    NoPConj   : PConj ;
    PConjConj : Conj -> PConj ;           -- and

                                             и
```

The vocative is optional. Any noun phrase can be made into vocative, which may be overgenerating (e.g. *I*).

```
    NoVoc   : Voc ;

    VocNP   : NP -> Voc ;                 -- my friend

                                             мой друг

  }
```

10.2 Russian Implementation

Russian implementation of `Phrase` API (concrete syntax for Russian). The functions below are considered straightforward.

```
concrete PhraseRus of Phrase = CatRus ** open Prelude, ResRus in {

  lin
    PhrUtt pconj utt voc = {s = pconj.s ++ utt.s ++ voc.s} ;

    UttS s = s ;
    UttQS qs = {s = qs.s ! QDir} ;
    UttImpSg pol imp = {s = pol.s ++ imp.s ! pol.p ! Masc! Sg} ;
    UttImpPl pol imp = {s = pol.s ++ imp.s ! pol.p ! Masc! Pl} ;

    UttIP ip = {s = ip.s ! PF Nom No NonPoss} ; --- Acc also
    UttIAdv iadv = iadv ;
    UttNP np = {s = np.s ! PF Acc No NonPoss} ;
    UttVP vp = {s = vp.s ! ClInfinit ! ASg Masc! P3} ;
    UttAdv adv = adv ;

    NoPConj = {s = []} ;
    PConjConj conj = conj ;

    NoVoc = {s = []} ;
    VocNP np = {s = "," ++ np.s ! PF Nom No NonPoss} ;
}
```

11 Question

11.1 Abstract API

Language-independent functions (abstract syntax) for forming questions.

```
abstract Question = Cat ** {
```

A question can be formed from a clause ('yes-no question') or with an interrogative.

```
fun
  QuestCl     : Cl -> QCl ;                  -- does John walk

                                                Иван идёт

  QuestVP     : IP -> VP -> QCl ;            -- who walks
```

```
                                                кто идёт
  QuestSlash  : IP -> Slash -> QCl ;          -- who does John love
                                                кого Иван любит
  QuestIAdv   : IAdv -> Cl -> QCl ;           -- why does John walk
                                                почему Иван идёт
  QuestIComp  : IComp -> NP -> QCl ;          -- where is John
                                                где Иван
```

Interrogative pronouns can be formed with interrogative determiners.

```
  IDetCN  : IDet -> Num -> Ord -> CN -> IP;  -- which five best songs
                                                какие пять лучших
                                                   песен
  AdvIP   : IP -> Adv -> IP ;                -- who in Europe
                                                кто в Европе
  PrepIP  : Prep -> IP -> IAdv ;             -- with whom
                                                с кем
  CompIAdv : IAdv -> IComp ;                 -- where
                                                где
```

More `IP`, `IDet`, and `IAdv` are defined in `Structural`.

```
  }
```

11.2 Russian Implementation

Russian implementation of `Question` API (concrete syntax for Russian).

```
concrete QuestionRus of Question = CatRus ** open ResRus,
 Prelude in {

 flags optimize=all_subs ;
 lin

   QuestCl cl = {s = \\b,cf,_ => cl.s ! b ! cf  } ;

   QuestVP kto spit =
      {s = \\b,clf,qf  => (predVerbPhrase kto spit).s!b!clf  } ;
```

Interrogative pronoun `Kto` is inflected according to the verb phrase's case (`yaGovoru.c`):

```
QuestSlash Kto yaGovoruO =
   let {  kom = Kto.s ! (mkPronForm yaGovoruO.c No NonPoss) ;
   o = yaGovoruO.s2 } in
   {s =  \\b,clf,_ => o ++ kom ++ yaGovoruO.s ! b ! clf
   } ;

QuestIAdv kak tuPozhivaesh =
   {s = \\b,clf,q => kak.s ++ tuPozhivaesh.s!b!clf } ;

QuestIComp pochemu stul =
   {s = \\b,clf,q => let ne = case b of

    {Pos =>[]; Neg => "не"}

     in
    pochemu.s ++ ne ++ stul.s! PF Nom No NonPoss } ;

PrepIP p ip = {s = p.s ++ ip.s ! PF Nom No NonPoss} ;
```

All parameter values: number (`n`), person (`p`), gender (`g`), animacy (`anim`) are inherited from interrogative pronoun (`ip`):

```
AdvIP ip adv = {
     s = \\c => ip.s ! c ++ adv.s ;
      n = ip.n; p=ip.p; g=ip.g; anim=ip.anim; pron=ip.pron
     } ;
```

Number (`n`) and gender (`g`) are inherited from interrogative determiner (`kakoj`). Animacy (`anim`) is inherited from common noun (`okhotnik`). The difference between Nominative (`Nom`) and other cases of interrogative determiner (`kakoj.c`) is in the choice of case for common noun (`okhotnik.s`), i.e. common noun agrees with interrogative determiner. Numeral (`pyat.s`) and ordinal (`umeluj.s`) agree with common noun (`okhotnik.s`):

```
IDetCN kakoj pyat umeluj okhotnik =
   {s = \\pf => case kakoj.c of {
      Nom =>
       kakoj.s ! AF (extCase pf) okhotnik.anim
        (gNum okhotnik.g kakoj.n) ++  pyat.s!
        (extCase pf) ! okhotnik.g ++ umeluj.s!
         AF (extCase pf) okhotnik.anim
         gNum okhotnik.g kakoj.n)++
         okhotnik.s ! kakoj.n ! (extCase pf) ;
```

```
        _ =>
         kakoj.s ! AF (extCase pf) okhotnik.anim
         (gNum okhotnik.g kakoj.n) ++
         pyat.s! (extCase pf) ! okhotnik.g ++
         umeluj.s!AF (extCase pf) okhotnik.anim
         (gNum okhotnik.g kakoj.n)++
         okhotnik.s ! kakoj.n ! kakoj.c };
       n = kakoj.n ;
       p = P3 ;
       pron = False;
       g = kakoj.g ;
       anim = okhotnik.anim
      } ;

   CompIAdv a = a ;
}
```

12 Relative

12.1 Abstract API

Language-independent functions (abstract syntax) for forming relative phrases.

```
abstract Relative = Cat ** {
  fun
```

The simplest way to form a relative clause is from a clause by a pronoun similar to *such that*.

```
    RelCl    : Cl -> RCl ;            -- such that John loves her
```

так что Иван её любит

The more proper ways are from a verb phrase (formed in `Verb`) or a sentence with a missing noun phrase (formed in `Sentence`).

```
    RelVP    : RP -> VP -> RCl ;      -- who loves John
```

кто любит Ивана

```
    RelSlash : RP -> Slash -> RCl ;   -- whom John loves
```

кого любит Иван

Relative pronouns are formed from an 'identity element' by prefixing or suffixing (depending on language) prepositional phrases.

```
    IdRP  : RP ;                      -- which
```

```
                                          который
      FunRP : Prep -> NP -> RP -> RP ;  -- all the roots of which
                                          все корни которого
  }
```

12.2 Russian Implementation

Russian implementation of `Relative` API (concrete syntax for Russian).

```
concrete RelativeRus of Relative = CatRus ** open ResRus,
 MorphoRus in {

  flags optimize=all_subs ; coding=utf8 ;

  lin
```

Такой (`takoj`) and *что* in Russian correspond to *such* and *that* in English, respectively:

```
   RelCl A =   {s = \\b,clf,gn,c, anim =>

    takoj.s !  AF c anim gn ++ "что" ++ A.s !  b !  clf};

   RelVP kotoruj gulyaet = { s = \\b,clf,gn, c, anim =>
      let { nu = numGNum gn } in
     kotoruj.s ! gn ! c ! anim ++ gulyaet.s2 ++
     gulyaet.s ! clf ! gn !P3 ++
     gulyaet.s3 ! genGNum gn ! nu
   } ;
```

Relative pronoun (`kotoruj`) agrees with slash in case (`yaVizhu.c`):

```
   RelSlash kotoruj yaVizhu = {s = \\b,clf,gn, _ ,
     anim => yaVizhu.s2 ++ kotoruj.s ! gn ! yaVizhu.c ! anim
            ++ yaVizhu.s!b!clf
   } ;

   FunRP p mama kotoruj = {s = \\gn,c, anim =>
     mama.s ! PF c No NonPoss ++ p.s ++ kotoruj.s !  gn ! p.c ! anim
   } ;

   IdRP = {s  = \\gn, c, anim => kotorujDet.s ! (AF c anim gn )} ;
}
```

13 Sentence

13.1 Abstract API

Language-independent functions (abstract syntax) for forming sentences.

```
abstract Sentence = Cat ** {
```

Clauses

The `NP VP` predication rule form a clause whose linearization gives a table of all tense variants, positive and negative. Clauses are converted to `S` (with fixed tense) in `Tensed`.

```
fun
  PredVP    : NP -> VP -> Cl ;          -- John walks
```

Иван идёт

Using an embedded sentence as a subject is treated separately. This can be overgenerating. E.g. *whether you go* as subject is only meaningful for some verb phrases.

```
  PredSCVP  : SC -> VP -> Cl ;          -- that you go makes me happy
```

что ты придёшь делает меня счастливым

Clauses missing object noun phrases

This category is a variant of the 'slash category' `S/NP` of GPSG [11] and categorial grammars, which in turn replaces movement transformations in the formation of questions and relative clauses. Except `SlashV2`, the construction rules can be seen as special cases of function composition, in the style of CCG [35]. **Note**: the set is not complete and lacks e.g. verbs with more than 2 places.

```
  SlashV2   : NP -> V2 -> Slash ;       -- (whom) he sees
```

(кого) он видит

```
  SlashVVV2 : NP -> VV -> V2 -> Slash; -- (whom) he wants to see
```

(кого) он хочет видеть

```
  AdvSlash  : Slash -> Adv -> Slash ;   -- (whom) he sees tomorrow
```

(кого) он увидит завтра

```
  SlashPrep : Cl -> Prep -> Slash ;     -- (with whom) he walks
```

(с кем) он гуляет

Imperatives

An imperative is straightforwardly formed from a verb phrase. It has variation over positive and negative, singular and plural. To fix these parameters, see `Phrase`.

```
  ImpVP     : VP -> Imp ;              -- go

                                          иди
```

Embedded sentences

Sentences, questions, and infinitival phrases can be used as subjects and (adverbial) complements.

```
  EmbedS    : S  -> SC ;               -- that you go

                                          что ты идёшь

  EmbedQS   : QS -> SC ;               -- whether you go

                                          идёшь ли ты

  EmbedVP   : VP -> SC ;               -- to go

                                          идти
```

Sentences

These are the 2 x 4 x 4 = 16 forms generated by different combinations of tense, polarity, and anteriority, which are defined in `Tense`.

```
fun
  UseCl  : Tense -> Ant -> Pol -> Cl  -> S ;

  UseQCl : Tense -> Ant -> Pol -> QCl -> QS ;

  UseRCl : Tense -> Ant -> Pol -> RCl -> RS ;

  }
```

13.2 Russian Implementation

Russian implementation of `Sentence` API (concrete syntax for Russian).

```
concrete SentenceRus of Sentence = CatRus ** open Prelude, ResRus
in {
  flags optimize=all_subs ; coding=utf8 ;
  lin
```

Verb (`vizhu`) and complement (`tebya`) agree with subject (`Ya`) in gender (`g`), number (`n`) and person (`p`):

```
   PredVP Ya tebyaNeVizhu = { s = \\b,clf =>
         let  {
           ya = Ya.s ! (case clf of {
                ClInfinit => (mkPronForm Acc No NonPoss);
                 _ => (mkPronForm Nom No NonPoss)
                 });

           ne = case b of {Pos=>""; Neg=>"не"};

           vizhu = tebyaNeVizhu.s ! clf ! (pgNum Ya.g Ya.n)! Ya.p;
           tebya = tebyaNeVizhu.s3 ! (pgen2gen Ya.g) ! Ya.n
           khorosho = tebyaNeVizhu.s2 ;
         }
         in if_then_else Str tebyaNeVizhu.negBefore
          (ya ++ ne ++ vizhu ++ tebya ++ khorosho)
          (ya ++ vizhu ++ ne ++ tebya ++ khorosho)
   } ;
```

Embedded sentence as a subject is assumed to be singular (`Sg`), third person (`P3`):

```
   PredSCVP sc vp = { s = \\b,clf =>
     let  {

           ne = case b of {Pos=>""; Neg=>"не"};

           vizhu = vp.s ! clf ! (ASg Neut)! P3;
           tebya = vp.s3 ! Neut ! Sg
         }
         in
     if_then_else Str vp.negBefore
          (sc.s ++ ne ++ vizhu ++ tebya)
          (sc.s ++ vizhu ++ ne ++ tebya)
   } ;
```

Verb (lubit) agrees with subject (ivan) in gender (g), number (n) and person (p). Complement (s2 and c) in the next four functions is inherited from the verb:

```
SlashV2 ivan lubit = { s=\\b,clf => ivan.s ! PF Nom No NonPoss ++
  lubit.s! (getActVerbForm clf (pgen2gen ivan.g) ivan.n ivan.p) ;
     s2=lubit.s2; c=lubit.c };

SlashVVV2 ivan khotet lubit =
  { s=\\b,clf => ivan.s ! PF Nom No NonPoss ++
      khotet.s! (getActVerbForm clf (pgen2gen ivan.g) ivan.n ivan.p)
      ++ lubit.s! VFORM Act VINF ;
    s2=lubit.s2;
    c=lubit.c
};

AdvSlash slash adv = {
    s  = \\b,clf => slash.s ! b ! clf ++ adv.s ;
    c = slash.c;
    s2 = slash.s2;
} ;

SlashPrep cl p =  {s=cl.s; s2=p.s; c=p.c} ;

ImpVP inf = {s = \\pol, g,n =>
  let
    dont  = case pol of {

               Neg => "не" ;

             _ => []
            }
       in
       dont ++ inf.s ! ClImper ! (gNum g n )!P2++
       inf.s2++inf.s3!g!n
} ;
```

Что in Russian corresponds to *that* in English:

```
      EmbedS s = {s = "что" ++ s.s} ;
```

In Russian "Whether you go" transformed into "go whether you":

```
EmbedQS qs = {s = qs.s ! QIndir} ;

EmbedVP vp = {s = vp.s2  ++ vp.s!ClInfinit!(ASg Masc) !P3
```

```
        ++ vp.s3!Masc!Sg} ;

   UseCl  t a p cl = {s = case t.t of {
        Cond => cl.s! p.p ! ClCondit ;
        _ => cl.s! p.p ! ClIndic (getTense t.t) a.a}};

   UseQCl t a p qcl= {s = case t.t of {
        Cond => qcl.s! p.p ! ClCondit ;
        _ => qcl.s!p.p! ClIndic (getTense t.t) a.a }};

   UseRCl t a p rcl ={s = \\gn,c,anim => case t.t of {
        Cond => [", "] ++ rcl.s! p.p ! ClCondit ! gn !c !anim ;
        _ => [", "] ++ rcl.s! p.p ! ClIndic (getTense t.t)
             a.a !gn !c !anim}};
}
```

14 Verb

14.1 Abstract API

Language-independent functions (abstract syntax) for forming verb phrases.

```
  abstract Verb = Cat ** {
```

Complementization rules

Verb phrases are constructed from verbs by providing their complements. There is one rule for each verb category.

```
    fun
      UseV     : V   -> VP ;              -- sleep

                                             спит

      ComplV2  : V2  -> NP -> VP ;        -- use it

                                             используй это

      ComplV3  : V3  -> NP -> NP -> VP ;  -- send a message to her

                                             пошли ей письмо

      ComplVV  : VV  -> VP -> VP ;        -- want to run

                                             хочет бежать
```

```
ComplVS  : VS  -> S  -> VP ;        -- know that she runs
                                       знаю, что она бегает
ComplVQ  : VQ  -> QS -> VP ;        -- ask if she runs
                                       спросить бегает ли она
ComplVA  : VA  -> AP -> VP ;        -- look red
                                       выглядит красным
ComplV2A : V2A -> NP -> AP -> VP ;  -- paint the house red
                                       покрасить дом красным
```

Other ways of forming verb phrases

Verb phrases can also be constructed reflexively and from copula-preceded complements.

```
ReflV2   : V2 -> VP ;               -- use itself
                                       использовать себя
UseComp  : Comp -> VP ;             -- be warm
                                       быть тёплым
```

Passivization of two-place verbs is another way to use them. In many languages, the result is a participle that is used as complement to a copula (*is used*), but other auxiliary verbs are possible (Ger. *wird angewendet*, It. *viene usato*), as well as special verb forms (Fin. *käytetään*, Swe. *används*, Rus: *используемСЯ*).

Note. the rule can be overgenerating, since the `V2` need not take a direct object.

```
PassV2   : V2 -> VP ;               -- be used
                                       быть использованным
```

Adverbs can be added to verb phrases. Many languages make a distinction between adverbs that are attached in the end vs. next to (or before) the verb.

```
AdvVP    : VP -> Adv -> VP ;        -- sleep here
                                       спит тут
AdVVP    : AdV -> VP -> VP ;        -- always sleep
                                       всегда спит
```

Agents of passives are constructed as adverbs with the preposition.

Complements to copula

Adjectival phrases, noun phrases, and adverbs can be used.

```
      CompAP   : AP  -> Comp ;            -- (be) small
                                             (быть) маленьким
      CompNP   : NP  -> Comp ;            -- (be) a soldier
                                             (быть) солдатом
      CompAdv  : Adv -> Comp ;            -- (be) here
                                             (быть) здесь
```

Coercions

Verbs can change subcategorization patterns in systematic ways, but this is very much language-dependent. The following two work in all the languages we cover.

```
      UseVQ   : VQ -> V2 ;                -- ask (a question)
                                             спросить (вопрос)
      UseVS   : VS -> V2 ;                -- know (a secret)
                                             знать (секрет)

  }
```

14.2 Russian Implementation

Russian implementation of `Verb` API (concrete syntax for Russian).

```
concrete VerbRus of Verb = CatRus ** open ResRus, Prelude in {
 flags optimize=all_subs ;  coding=utf8 ;
  lin
```

In the next three rules numerous cases correspond to the inflection forms of copula verb *to be* (быть in Russian), which is imperfective in aspect(`asp`), active in voice (`w`) with prefix negation (`negBefore`). Complements (`s2, s3`) are empty:

```
    CompNP masha = { s=\\clf,gn,p => case clf of {
     (ClIndic Present _) =>  masha.s ! (mkPronForm Nom No NonPoss) ;
     (ClIndic Past _) => case  gn of      {
```

```
    (ASg Fem) => "была"++masha.s !  (mkPronForm Inst No NonPoss);
    (ASg Masc) => "был" ++ masha.s!(mkPronForm Inst No NonPoss);
    (ASg Neut) => "было" ++ masha.s!(mkPronForm Inst No NonPoss);
    APl => "были" ++ masha.s !  (mkPronForm Inst No NonPoss)

  };
   (ClIndic Future _) => case gn of{
      APl => case p of {

     P3 => "будут"++masha.s !  (mkPronForm Inst No NonPoss);
     P2 => "будете"++masha.s !(mkPronForm Inst No NonPoss);
     P1 => "будем"++masha.s !  (mkPronForm Inst No NonPoss)

     };
     (ASg _) => case p of {

     P3=>"будет"++masha.s!(mkPronForm Inst No NonPoss) ;
     P2 => "будешь"++ masha.s !  (mkPronForm Inst No NonPoss) ;
     P1=> "буду"++ masha.s !  (mkPronForm Inst No NonPoss)

       } --case p
   }; --case gn
   ClCondit => "" ;
   ClImper  => case (numGNum gn) of {

     Sg => "будь" ++ masha.s !  (mkPronForm Inst No NonPoss);
     Pl => "будьте" ++ masha.s !  (mkPronForm Inst No NonPoss)};
    ClInfin => "быть" ++ masha.s !  (mkPronForm Inst No NonPoss)

   };  -- case clf
   asp = Imperfective ;
   w = Act;
   negBefore = True;
   s2 = "";
   s3 = \\g,n => ""
 } ;
```

Unlike the previous function here not only copular inflects, but also adjective (`zloj`) inflects in number and gender:

```
CompAP zloj ={ s= \\clf,gn,p => case clf of {-- person is ignored

    ClInfinit => "быть" ++ zloj.s!AF Inst Animate (ASg Masc) ;

   ClImper => case gn of {
```

```
    (ASg _) => "будь" ++ zloj.s !  AF Inst Animate (ASg Masc);
    APl => "будьте" ++ zloj.s !  AF Inst Animate APl };
```

Infinitive does not save `GenNum`, but indicative does for the sake of adjectival predication:

```
ClIndic Present _ =>  zloj.s ! AF Nom Animate gn ;
ClIndic Past _ => case gn of {

  (ASg Fem) => "была" ++ zloj.s!  AF Nom Animate (ASg Fem);
  (ASg Masc) => "был" ++ zloj.s!  AF Nom Animate (ASg Masc);
  (ASg Neut) => "было" ++ zloj.s!  AF Nom Animate (ASg Neut);
  APl => "были" ++ zloj.s!  AF Nom Animate APl };

ClIndic Future _ => case gn of
  { APl => case p of {

  P3 => "будут" ++ zloj.s!  AF Nom Animate APl;
  P2 => "будете" ++ zloj.s!  AF Nom Animate APl;
  P1 => "будем" ++ zloj.s!  AF Nom Animate APl

     } ;
 (ASg _) => case p of

  P3 => "будет" ++ zloj.s!  AF Nom Animate (ASg (genGNum gn));
  P2 => "будешь"++ zloj.s!  AF Nom Animate (ASg (genGNum gn));
  P1=> "буду" ++ zloj.s!  AF Nom Animate (ASg (genGNum gn))

    }
  };
 ClCondit => ""
} ;
asp = Imperfective ;
w = Act;
negBefore = True;
s2 = "";
s3 = \\g,n=> ""
} ;
```

Verb phrases can also be formed from common nouns (*человек*) [*a man*], noun phrases (*самый молодой*) [*the youngest*] and adjectives (*молод*) [*young*]. The second rule is overgenerating: *каждый человек* [*every man*] has to be ruled out on semantic grounds.

Note: we omit a dash "–" because it will cause problems with negation word order: *Я – не волшебник* (*I am not a wizard*). Alternatively, we can consider verb-based VP and all the rest.

```
CompAdv zloj = { s= \\clf,gn,p => case clf of {
   ClImper => case gn of {
     ASg _ => "будь" ++ zloj.s; -- person is ignored
     APl => "будьте" ++ zloj.s };
   ClInfinit => "быть " ++ zloj.s;
   ClIndic Present  _ => zloj.s ;
   ClIndic Past _ => case gn of {
     (ASg Fem) => "была" ++ zloj.s;
     (ASg Masc) => "был" ++ zloj.s;
     (ASg Neut) => "было" ++ zloj.s;
     APl => "были" ++ zloj.s
     };
   ClIndic Future _ => case gn of {
     (ASg _) => "будет" ++ zloj.s;
     APl => "будут" ++ zloj.s };
   ClCondit => "" } ;
  asp = Imperfective ;
  w = Act;
  s2 = "";
  negBefore = True;
  s3 = \\g,n => ""
} ;

UseComp comp = comp ;
UseVS, UseVQ = \vv -> {s = vv.s ; asp = vv.asp; s2 = [] ;
   c = Acc} ;
```

In the following functions aspect parameter (`asp`) is inherited from the verb (`se, dat, v, vidit` etc.) active (`Act`) in voice (`w`); complements (`s2, s3`) are empty; negation is prefixed.
A simple verb can be made into a verb phrase with an empty complement. There are two versions, depending on if we want to negate the verb:

```
UseV se = {
   s=\\clf,gn,p =>
     se.s ! (getActVerbForm clf (genGNum gn) (numGNum gn) p) ;
   asp = se.asp ;
   w=Act;
   s2 = "";
   negBefore = True;
   s3 = table{_=> table{_ => ""}}
  } ;
```

The rule for using transitive verbs is the complementization rule:

```
ComplV2 se tu = {
  s =\\clf,gn,p =>
   se.s ! (getActVerbForm clf (genGNum gn) (numGNum gn) p)
   ++ se.s2 ++ tu.s ! (mkPronForm se.c No NonPoss) ;
  asp = se.asp ;
  w = Act;
  s2 = "";
  s3 = \\g,n => "";
  negBefore = True
} ;
```

`dat` verb requires a certain case (`dat.c`) from the complement (`dat.c` from `tu.s` and `dat.c2` from `pivo.s`):

```
ComplV3 dat tu pivo = let tebepivo =
   dat.s2 ++ tu.s ! PF dat.c No NonPoss ++ dat.s4 ++
   pivo.s ! PF dat.c2 Yes NonPoss
    in
   {s  = \\clf,gn,p => dat.s !
     (getActVerbForm clf (genGNum gn) (numGNum gn) p) ++
     tebepivo ;
    asp = dat.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
    s3 = \\g,n=> ""
} ;
```

Self in English corresponds to *себя* (`sebya`) in Russian:

```
ReflV2 v = {
    s  = \\clf,gn,p => v.s !
     (getActVerbForm clf (genGNum gn) (numGNum gn) p) ++
      v.s2 ++ sebya!v.c;
    asp = v.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
    s3 = \\g,n=> ""
} ;
```

To generate *сказал, что Иван гуляет / не сказал, что Иван гуляет* (*told that Ivan walks / did not tell that Ivan walks*):

```
ComplVS vidit tuUlubaeshsya = {
  s = \\clf,gn,p => vidit.s !
  (getActVerbForm clf (genGNum gn) (numGNum gn) p)


   ++ [", что"] ++ tuUlubaeshsya.s ;

  asp = vidit.asp;
  w = Act;
  s2="";
  negBefore = True;
  s3 = \\g,n => ""
} ;
```

To generate *может гулять; не пытается работать* (*can walk; does not try to work*):

```
ComplVV putatsya bezhat = {
    s = \\clf,gn,p =>
     putatsya.s ! (getActVerbForm clf (genGNum gn)
     (numGNum gn) p) ++ bezhat.s!ClInfinit !gn!p ;
    asp = putatsya.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
    s3 =\\g,n => ""
} ;
ComplVQ dat esliOnPridet = {
    s = \\clf,gn,p =>
      dat.s ! (getActVerbForm clf (genGNum gn) (numGNum gn) p)
      ++ esliOnPridet.s ! QDir ;
    asp = dat.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
    s3 = \\g,n=> ""
} ;
ComplVA vuglyadet molodoj = {
    s = \\clf,gn,p => vuglyadet.s!
      (getActVerbForm clf (genGNum gn) (numGNum gn) p) ;
    asp = vuglyadet.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
```

```
    s3 = \\g,n => molodoj.s!(AF Inst Animate (gNum g n))
} ;
```

Verb requires a certain case (`obechat.c`) from noun phrase (`tu`). Adjective's (`molodoj.s`) form depends on noun phrase's animacy (`tu.anim`):

```
ComplV2A obechat tu molodoj = {
    s = \\clf,gn,p => obechat.s2++obechat.s !
      (getActVerbForm clf (genGNum gn) (numGNum gn) p)
      ++  tu.s ! PF obechat.c No NonPoss ++
      molodoj.s!AF Inst tu.anim (pgNum tu.g tu.n) ;
    asp = obechat.asp ;
    w = Act;
    negBefore = True;
    s2 = "";
    s3 = \\g,n =>""
} ;
```

The difference between the next two functions is the word order (prefix of postfix adverb) in the complement (`s2`). All parameters are inherited from the verb argument (`poet`):

```
AdvVP poet khorosho = {
´   s = \\clf,gn,p => poet.s ! clf!gn!p;
    s2 = poet.s2 ++ khorosho.s;
    s3 = poet.s3;
    asp = poet.asp;
    w = poet.w;
    t = poet.t ;
    negBefore = poet.negBefore } ;

AdVVP khorosho poet = {
    s = \\clf,gn,p => poet.s ! clf!gn!p;
    s2 = khorosho.s ++ poet.s2;
    s3 = poet.s3;
    asp = poet.asp;
    w = poet.w;
    t = poet.t ;
    negBefore = poet.negBefore } ;

PassV2 se = {s=\\clf,gn,p =>
   se.s ! (getPassVerbForm clf (genGNum gn) (numGNum gn) p) ;
```

```
      asp=se.asp;
      w=Pass;
      s2 = se.s2;
      negBefore = True;
      s3 = table{_=> table{_ => ""}}
  };
}
```

15 Paradigms

This is an API for the user of the Russian resource grammar for adding lexical items. It gives functions for forming expressions of open categories: nouns, adjectives, verbs.

Closed categories (determiners, pronouns, conjunctions) are accessed through the resource syntax API, `Structural`.

The implementations of paradigms are placed in lower-level module `MorphoRus`, which is not shown, because it is too big (around 2000 lines) and too detailed. Also our work here is concentrated on syntax and not on morphology.

The main difference with `MorphoRus` is that the types referred to are compiled resource grammar types. We have moreover had the design principle of always having existing forms, rather than stems, as string arguments of the paradigms.

The structure of functions for each word class `C` is the following: first we give a handful of patterns that aim to cover all regular cases. Then we give a worst-case function `mkC`, which serves as an escape to construct the most irregular words of type `C`.

```
resource ParadigmsRus = open
  (Predef=Predef),
  Prelude,
  MorphoRus,
  CatRus,
  NounRus
  in {

flags  coding=utf8 ;
```

15.1 Parameters

To abstract over gender names, we define the following identifiers.

```
oper
  Gender : Type ;
```

```
masculine : Gender ;
feminine  : Gender ;
neuter    : Gender ;
```

To abstract over case names, we define the following.

```
Case : Type ;

nominative    : Case ;
genitive      : Case ;
dative        : Case ;
accusative    : Case ;
instructive   : Case ;
prepositional : Case ;
```

In some (written in English) textbooks accusative case is put on the second place. However, we follow the case order standard for Russian textbooks. To abstract over number names, we define the following.

```
Number : Type ;

singular : Number ;
plural   : Number ;

Animacy: Type ;

animate: Animacy;
inanimate: Animacy;
```

15.2 Nouns

Best case: indeclinabe nouns: *кофе* (*coffee*), *пальто* (*coat*), *ВУЗ* (*university*).

```
mkIndeclinableNoun: Str -> Gender -> Animacy -> N ;
```

Worst case – give six singular forms: Nominative, Genitive, Dative, Accusative, Instructive and Prepositional; corresponding six plural forms and the gender. May be the number of forms needed can be reduced, but this requires a separate investigation. Animacy parameter (determining whether the Accusative form is equal to the Nominative or the Genitive one) is actually of no help, since there are a lot of exceptions and the gain is just one form less.

```
mkN  : (nomSg, genSg, datSg, accSg, instSg, preposSg,
        nomPl, genPl, datPl, accPl, instPl, preposPl: Str)
        -> Gender -> Animacy -> N ;
```

(*мужчина, мужчину, мужчине, мужчину, мужчиной, мужчине мужчины, мужчин, мужчинам, мужчин, мужчинами, мужчинах*) (*man*)

The regular function captures the variants for some popular nouns endings from the list below:

```
regN              : Str -> N ;
```

Here are some common patterns. The list is far from complete.

Feminine patterns

feminine, inanimate, ending with *а*, instructive case – *машинОЙ* (*car*):

```
nMashina          : Str -> N ;
```

feminine, inanimate, ending with *а*, instructive case – *единицЕЙ* (*one*):

```
nEdinica          : Str -> N ;
```

feminine, animate, ending with *а*, nominative case – *женщина (woman)*:

```
nZhenchina        : Str -> N ;
```

feminine, inanimate, ending with *га_ка_ха*, nominative case – *нога* (*leg*):

```
nNoga             : Str -> N ;
```

feminine, inanimate, ending with *ия*, nominative case – *малярия* (*malaria*):

```
nMalyariya        : Str -> N ;
```

feminine, animate, ending with *я*, nominative case – *тётя* (*aunt*):

```
nTetya            : Str -> N ;
```

feminine, inanimate, ending with *ь*(soft sign), nominative case – *боль* (*pain*):

```
nBol              : Str -> N ;
```

Neuter patterns

neutral, inanimate, ending with *ее*, nominative case – *обезболивающее* (*painkiller*):

```
nObezbolivauchee : Str -> N ;
```

neutral, inanimate, ending with *е*, nominative case – *произведение* (*product*):

```
nProizvedenie     : Str -> N ;
```

neutral, inanimate, ending with *о*, nominative case – *число* (*number*):

```
nChislo           : Str -> N ;
```

neutral, inanimate, ending with *ое*, nominative case – *животное* (*animal*):

```
nZhivotnoe        : Str -> N ;
```

Masculine patterns

Ending with consonant:
masculine, inanimate, ending with *ел*, genitive case – *пепЛА* (*ash's*):

```
nPepel          : Str -> N ;
```

animate, plural – *братЬЯ* (*brothers*):

```
nBrat           : Str -> N ;
```

same as above, but inanimate, nominative case – *стул* (*chair*):

```
nStul           : Str -> N ;
```

plural genitive – *малышЕЙ* (*babies'*):

```
nMalush         : Str -> N ;
```

genitive case – *потолКА* (*ceiling's*)

```
nPotolok        : Str -> N ;
```

The next four differ in plural nominative and/or accusative form(s):
plural – *банкИ* (*banks*) (`Nom=Acc`):

```
nBank           : Str -> N ;
```

same as above, but animate, nominative case – *стоматолог* (*dentist*):

```
nStomatolog     : Str -> N ;
```

genitive case – *адресА* (*address'*) (`Nom=Acc`):

```
nAdres          : Str -> N ;
```

plural – *телефонЫ* (*phones*) (`Nom=Acc`):

```
nTelefon        : Str -> N ;
```

masculine, inanimate, ending with *ь* (soft sign), nominative case – *ноль* (*zero*):

```
nNol            : Str -> N ;
```

masculine, inanimate, ending with *ень*, nominative case – *уровень* (*level*):

```
nUroven         : Str -> N ;
```

Nouns used as functions need a preposition. The most common is with genitive.

```
mkFun           : N -> Prep -> N2 ;
mkN2            : N -> N2 ;
mkN3            : N -> Prep -> Prep -> N3 ;
```

Proper names

Иван, Маша:

```
mkPN              : Str -> Gender -> Animacy -> PN ;

nounPN            : N -> PN ;
```

On the top level, it is maybe `CN` that is used rather than `N`, and `NP` rather than `PN`.

```
mkCN              : N -> CN ;
mkNP              : Str -> Gender -> Animacy -> NP ;
```

15.3 Adjectives

Non-comparison (only positive degree) one-place adjectives need 28 (4 by 7) forms in the worst case:

(Masculine — Feminine — Neutral — Plural) *

(Nominative — Genitive — Dative — Accusative Inanimate — Accusative Animate — Instructive — Prepositional).

Notice that 4 short forms, which exist for some adjectives are not included in the current description, otherwise there would be 32 forms for positive degree.

The regular function captures the variants for some popular adjective endings below. The first string argument is the masculine singular form, the second is comparative:

```
regA              : Str -> Str -> A ;
```

Invariable adjective is a special case: *хаки* (*khaki*) , *мини* (*mini*), *хинди* (*Hindi*), *нетто* (*netto*):

```
adjInvar          : Str -> A ;
```

Some regular patterns depending on the ending.
ending with *ый*, nominative case, masculine – *старый* (*old*):

```
AStaruyj          : Str -> Str -> A ;
```

ending with *ий*, masculine, genitive case – *маленькОГО* (*small*):

```
AMalenkij         : Str -> Str -> A ;
```

ending with *ий*, masculine, genitive case – *хорошЕГО* (*good*):

```
AKhoroshij        : Str -> Str -> A ;
```

ending with *ой*, plural – *молодЫЕ* (*young*) :

```
AMolodoj          : Str -> Str -> A ;
```

ending with *ой*, plural – *какИЕ* (*which*):

```
AKakoj_Nibud      : Str -> Str -> Str -> A ;
```

Two-place adjectives need a preposition and a case as extra arguments. *делим на* (*divisible by*):

```
mkA2              : A -> Str -> Case -> A2 ;
```

Comparison adjectives need a positive adjective (28 forms without short forms). Taking only one comparative form (non-syntactic) and only one superlative form (syntactic) we can produce the comparison adjective with only one extra argument - non-syntactic comparative form. Syntactic forms are based on the positive forms.

```
mkADeg            : A -> Str -> ADeg ;
```

On top level, there are adjectival phrases. The most common case is just to use a one-place adjective.

```
ap                : A  -> IsPostfixAdj -> AP ;
```

15.4 Adverbs

Adverbs are not inflected.

```
mkAdv             : Str -> Adv ;
```

15.5 Verbs

In our lexicon description (`Verbum`) there are 62 forms:

2 (Voice) *
{ 1 (infinitive)
+
[2(Number) * 3(Person)](imperative)
+
[[2(Number) * 3(Person)](present)
+
[2(Number) * 3(Person)](future)
+
4(GenNum)(past)](indicative)
+
4 (GenNum) (subjunctive) }.

Participles (present and past) and gerund forms are not included, since they function more like adjectives and adverbs respectively rather than verbs (see also 3.2). Such separation is, however, non-standard and is not present in the GF resource grammars for other languages. Aspect is regarded as an inherent parameter of a verb. Notice that some forms are never used for some verbs.

```
Voice: Type;
Aspect: Type;
Tense : Type;
Bool: Type;
Conjugation: Type ;
```

гуляЕШЬ, гуляЕМ (*walk*):

```
first: Conjugation;
```

Verbs with vowel *ё*: *даёшь* (*give*), *пьёшь* (*drink*):

```
firstE: Conjugation;
```

видИШЬ, видИМ (*see*):

```
second: Conjugation;
```

хочЕШЬ, хотИМ (*want*):

```
mixed: Conjugation;
```

irregular:

```
dolzhen: Conjugation;

true: Bool;
false: Bool;

active: Voice ;
passive: Voice ;
imperfective: Aspect;
perfective: Aspect ;
```

The worst case need 6 forms of the present tense in indicative mood: *я бегу* (*I run*), *ты бежишь* (*you run*), *он бежит* (*he runs*), *мы бежим* (*we run*), *вы бежите* (*you run*), *они бегут* (*they run*); a past form (singular, masculine: *я бежал* (*I run*)), an imperative form (singular, second person: *беги* (*run*)), an infinitive (*бежать* (*to run*)). Inherent aspect should also be specified.

```
mkV : Aspect -> (presentSgP1,presentSgP2,presentSgP3,
                        presentPlP1,presentPlP2,presentPlP3,
      pastSgMasculine,imperative,infinitive: Str) -> V ;
```

Common conjugation patterns are two conjugations: first – verbs ending with *ать/ять* and second – *ить/еть*. Instead of 6 present forms of the worst case, we only need a present stem and one ending (singular, first person): *я люблю* (*I love*), *я жду* (*I wait*), etc. To determine where the border between stem and ending lies it is sufficient to compare first person from with second person form: *я люблю* (*I love*), *ты любишь* (*you love*). Stems should be the same. So the definition for verb *любить* (*to love*) looks like: `regV Imperfective Second "люб" "лю" "любил" "люби" "любить";`

```
regV : Aspect -> Conjugation -> (stemPresentSgP1,
  endingPresentSgP1,pastSgP1,imperative,infinitive: Str) -> V ;
```

Two-place verbs, and the special case with direct object. Notice that a particle can be included in a `V`.
войти в дом (*come in into the house*), *в*, accusative:

```
mkV2              : V   -> Str -> Case -> V2 ;
```

сложить письмо в конверт (*put the letter into the envelope*):

```
mkV3              : V -> Str -> Str -> Case -> Case -> V3 ;
```

видеть (*to see*), *любить* (*to love*):

```
dirV2             : V -> V2 ;
tvDirDir          : V -> V3 ;
```

16 Automatically generated test examples

Automatically generated test examples of using the resource grammar library functions are intended for proof-reading and also reflect the coverage of the resource library. Below we show the test definitions together with their Russian and English linearizations. Corresponding linearizations can also be generated in other supported languages.

The purpose of the test is to cover all the syntactic resource library functions. Every function from the language-independent API is tested at least once in the test.

Of course, corresponding linearizations in different languages are not perfect translations, since they use exactly the same syntactic structures. The lexicon entries are taken from the basic interlingua lexicon.

16.1 Test definitions

```
--# -path=.:../abstract:../../prelude

abstract ResExamples = Lang ** {
fun
  ex1, ex2, ex4, ex8, ex13, ex19, ex20, ex23: Utt;
  ex3, ex5, ex6, ex7, ex10, ex12, ex14, ex15, ex16, ex24, ex25,
  ex26, ex27: S;
  ex9: Phr;
  ex11, ex17, ex18, ex21, ex22 : Text;

def
  ex1 = UttS (UseCl TPres ASimul PPos (PredVP (UsePron he_Pron)
          (AdvVP (UseV sing_V) (AdAdv almost_AdA (PositAdvAdj
           correct_A)))));

  ex2 = UttAdv (SubjS when_Subj (ConjS and_Conj (BaseS (UseCl
           TPast ASimul PPos (PredVP everybody_NP (UseComp
          (CompAP (ConjAP and_Conj (BaseAP (PositA young_A)
          (PositA beautiful_A)))))) (UseCl TPast ASimul PPos
          (PredVP everything_NP (ComplVA become_VA (PositA
           probable_AS)))))));

  ex3 = UseCl TPres ASimul PPos (CleftNP (PredetNP only_Predet
          (DetCN (DetPl (PlQuant IndefArt) (NumInt 2) NoOrd)
          (UseN woman_N))) (UseRCl TCond ASimul PPos (RelSlash
           IdRP (AdvSlash (SlashPrep (PredVP (UsePron i_Pron)
          (ComplVV want_VV (PassV2 see_V2))) with_Prep)
          (PrepNP in_Prep (DetCN (DetSg
          (SgQuant DefArt) NoOrd) (UseN rain_N)))))));


  ex4 = UttNP (DetCN someSg_Det (RelCN (UseN day_N) (UseRCl TFut
           ASimul PPos (RelCl (ExistNP (AdvNP (DetCN (DetSg
          (SgQuant IndefArt) NoOrd) (UseN peace_N))
          (PrepNP on_Prep (DetCN (DetSg (SgQuant IndefArt)
           NoOrd) (UseN earth_N)))))))));

  ex5 = UseCl TPres ASimul PPos (PredVP (UsePron they_Pron) (AdvVP
          (ProgrVP (UseV play_V)) (ComparAdvAdjS less_CAdv clever_A
          (UseCl TPres ASimul PPos (GenericCl (UseV think_V))))));
```

```
ex6 = UseCl TPres ASimul PPos (CleftAdv (AdvSC (EmbedVP (AdVVP
        always_AdV (UseV stop_V)))) (UseCl TPres ASimul PPos
       (PredVP (UsePron we_Pron)
       (ComplV2 beg_V2V (UsePron youPl_Pron)))));

ex7 = UseCl TCond ASimul PNeg (PredVP (UsePron i_Pron)
       (ComplV3 give_V3 (DetCN (DetPl (PlQuant IndefArt)
       (AdNum (AdnCAdv more_CAdv) (NumNumeral (num
       (pot2as3 (pot1as2 (pot0as1 (pot0 n3)))))) NoOrd)
       (UseN star_N)) (DetCN (DetSg (SgQuant this_Quant) NoOrd)
       (UseN restaurant_N))));

ex8 = UttImpSg PPos (ImpVP (ComplV2A paint_V2A (DetCN (DetSg
       (SgQuant DefArt) NoOrd) (UseN earth_N)) (DConjAP
        both7and_DConj (BaseAP (ComparA small_A (DetCN (DetSg
       (SgQuant DefArt) NoOrd) (UseN sun_N))) (ComparA big_A
       (DetCN (DetSg (SgQuant DefArt) NoOrd) (UseN moon_N )))))));

ex9 = UseCl TPres ASimul PPos (PredVP everybody_NP (ComplVQ
        wonder_VQ (UseQCl TPres ASimul PPos (QuestSlash
        whatSg_IP (SlashV2 (UsePron youSg_Pron)love_V2)))));

ex10 = UseCl TPres ASimul PPos (PredSCVP (EmbedS (UseCl TPres
         ASimul PNeg (PredVP (UsePron i_Pron) (UseComp
        (CompAP (ReflA2 married_A2))))))
        (ComplV2 kill_V2 (UsePron i_Pron)));

ex11 = TQuestMark (PhrUtt (PConjConj and_Conj) (UttQS (UseQCl
         TPres ASimul PNeg (QuestIAdv why_IAdv (PredVP
        (DetCN (DetSg MassDet NoOrd) (UseN art_N)) (UseComp
        (CompAP (ComparA (UseA2 easy_A2V) (DetCN (DetSg
         MassDet NoOrd) (UseN science_N))))))))) NoVoc) TEmpty;

ex12 = UseCl TPres ASimul PPos (CleftNP (DetCN (DetSg (SgQuant
         IndefArt) NoOrd) (UseN dog_N)) (UseRCl TPres ASimul
         PPos (RelSlash (FunRP with_Prep (DetCN (DetSg (SgQuant
         IndefArt) NoOrd) (UseN friend_N)) IdRP) (SlashVVV2
         (DetCN (DetSg (SgQuant (PossPron i_Pron)) NoOrd)
         (UseN2 brother_N2)) can_VV play_V2))));

ex13 = ImpPl1 (ComplVS hope_VS (DConjS either7or_DConj (BaseS
        (UseCl TPres ASimul PPos (PredVP (DetCN (DetSg
        (SgQuant DefArt) NoOrd) (ComplN2 father_N2 (DetCN
```

```
          (DetSg (SgQuant DefArt) NoOrd) (UseN baby_N))))
          (UseV run_V))) (UseCl TPres ASimul PPos (PredVP
          (DetCN (DetSg (SgQuant DefArt) NoOrd)(UseN3
           distance_N3)) (UseComp (CompAP (PositA small_A))))))));

ex14 = UseCl TPres ASimul PNeg (PredVP (UsePron i_Pron) (AdvVP
          (ReflV2 (UseVS fear_VS)) now_Adv));

ex15 = UseCl TPres ASimul PPos (PredVP (UsePron i_Pron) (ComplV2
          (UseVQ wonder_VQ) (ConjNP or_Conj
          (BaseNP somebody_NP something_NP))));

ex16 = UseCl TPres ASimul PPos (PredVP (DetCN every_Det
          (UseN baby_N)) (UseComp (CompNP (DConjNP either7or_DConj
          (BaseNP (DetCN (DetSg (SgQuant IndefArt) NoOrd)
          (UseN boy_N)) (DetCN (DetSg (SgQuant
           IndefArt) NoOrd) (UseN girl_N)))))));

ex17 = TQuestMark (PhrUtt NoPConj (UttQS (UseQCl TPres ASimul
           PPos (QuestVP (IDetCN whichSg_IDet NoNum NoOrd
          (ApposCN (ComplN2 (ComplN3 distance_N3 (DetCN (DetSg
          (SgQuant DefArt) NoOrd) (UseN house_N))) (DetCN (DetSg
          (SgQuant DefArt) NoOrd) (UseN bank_N))) (DetCN (DetSg
          (SgQuant DefArt) (OrdSuperl short_A)) (UseN road_N))))
          (PassV2 find_V2)))) NoVoc) TEmpty;

ex18 = TQuestMark (PhrUtt NoPConj ( UttQS (UseQCl TPres ASimul
           PPos (QuestIComp (CompIAdv where_IAdv) (DetCN (DetSg
          (SgQuant DefArt) NoOrd) (RelCN (UseN teacher_N)
          (UseRCl TPres ASimul PPos (RelVP IdRP (ComplV3 sell_V3
          (PPartNP (DetCN (DetPl (PlQuant DefArt) NoNum NoOrd)
          (UseN book_N)) read_V2) (DetCN (DetPl (PlQuant IndefArt)
           NoNum NoOrd) (UseN student_N)))))))))) NoVoc) TEmpty;

ex19 = UttIAdv (PrepIP with_Prep (AdvIP whoSg_IP (ConjAdv
           and_Conj (BaseAdv (PositAdvAdj cold_A)
          (PositAdvAdj warm_A)))));

ex20 = UttAdv (DConjAdv either7or_DConj (ConsAdv here7from_Adv
          (BaseAdv there_Adv everywhere_Adv)));

ex21 = TExclMark (PhrUtt NoPConj (UttImpPl PNeg (ImpVP
          (UseV die_V))) please_Voc) TEmpty;
```

```
  ex22 = TQuestMark (PhrUtt NoPConj (UttIP (IDetCN how8many_IDet
           NoNum NoOrd (UseN year_N))) (VocNP (DetCN (DetSg
          (SgQuant PossPron i_Pron)) NoOrd) (UseN friend_N))))
           TEmpty;

  ex23 = UttVP (PassV2 know_V2);

  ex24 = UseCl TPres ASimul PPos (PredVP (DetCN (DetSg MassDet
           NoOrd) (SentCN (UseN song_N) (EmbedVP (UseV sing_V))))
          (UseComp (CompAP (PositA (UseA2 easy_A2V)))));

  ex25 = UseCl TPast ASimul PNeg (PredVP (UsePron she_Pron)
          (ComplV2 know_V2 (DetCN (DetSg MassDet NoOrd) (AdvCN
          (UseN industry_N) (PrepNP before_Prep (DetCN (DetSg
          (SgQuant DefArt) NoOrd) (UseN university_N)))))));

 ex26 = UseCl TPres ASimul PPos (PredVP (UsePron she_Pron)
         (UseComp (CompAP (AdAP almost_AdA (SentAP (ComplA2
          married_A2 (DetCN (DetSg (SgQuant (PossPron she_Pron))
          NoOrd) (UseN cousin_N))) (EmbedQS (UseQCl TPast ASimul
          PPos (QuestCl (PredVP (UsePron youPol_Pron)
         (ComplV2 watch_V2 (DetCN (DetSg (SgQuant DefArt) NoOrd)
         (UseN television_N)))))))))));

  ex27 = UseCl TPres ASimul PPos (ImpersCl (ComplVV can8know_VV
           (UseComp (CompAdv (PositAdvAdj important_A)))));
}
```

16.2 English linearizations

Example 1.

He sings almost correctly

Example 2.

When everybody was young and beautiful and everything became probable

Example 3.

It is only 2 women that I would want to be seen in the rain with

Example 4.

Some day such that there will be a peace on an earth

Example 5.

They are playing less cleverly than one thinks

Example 6.

It is always to stop that we beg you

Example 7.

I wouldn't give more than three stars to this restaurant

Example 8.

Paint the earth both smaller than the sun and bigger than the moon

Example 9.

Everybody wonders what you love

Example 10.

That I am not married to myself kills me

Example 11.

And why isn't art easier than science?

Example 12.

It is a dog a friend with which my brother can play

Example 13.

Let's hope that either the father of the baby runs or the distance is small

Example 14.

I don't fear myself now

Example 15.

I wonder somebody or something

Example 16.

Every baby is either a boy or a girl

Example 17.

Which distance from the house to the bank the shortest road is found?

Example 18.

Where is the teacher that sells the books read to students?

Example 19.

With who coldly and warmly

Example 20.

Either from here, there or everywhere

Example 21.

Don't die please!

Example 22.

How many years, my friend?

Example 23.

To be known

Example 24.

Song to sing is easy

Example 25.

She didn't know industry before the university

Example 26.

She is almost married to her cousin if you watched the television

Example 27.

It can be importantly

16.3 Russian linearizations

Example 1.

он поёт почти правильно

Example 2.

когда все были молодые и красивые и всё стало возможным

Example 3.

это единственные 2 женщины, с которыми я хотел бы видеться в дожде

Example 4.

некоторый день, такой что будет существовать мир на земле

Example 5.

они играют менее умно чем ты думаешь

Example 6.

это всегда останавливать, мы просим вас

Example 7.

я не давал бы болеее чем три звезды этому ресторану

Example 8.

рисуй землю как меньше солнца, так и больше луны

Example 9.

все интересуют что вы любите

Example 10.

что я не замужем за собой убивает меня

Example 11.

и почему искусство не легче науки?

Example 12.

это собака, с другом с которой мой брат может играть

Example 13.

давайте будем надеять, что либо отец малыша бегает, либо расстояние маленькое

Example 14.

я не бою себя сейчас

Example 15.

я интересую кого-то или что-то

Example 16.

каждый малыш либо мальчик, либо девочка

Example 17.

которое расстояние от дома к банку короткая дорога находится?

Example 18.

где учитель, который продаёт книги студентам?

Example 19.

с кто холодно и тепло

Example 20.

либо отсюда, либо там, либо везде

Example 21.

не умирайте пожалуйста!

Example 22.

сколько годов, мой друг?

Example 23.

знаться

Example 24.

песня петь лёгкая

Example 25.

она не знала производство перед университетом

Example 26.

она почти замужем за её кузеном, вы смотрели телевидение

Example 27.

может быть важно

Technical report B

Syntax editing in GF

The editing procedure in GF is strongly connected to the concept of interactive theorem proof construction in proof editors like ALF[23] and Alfa[14].

Proof checker and user interface are the two main parts of such a proof editor, which are usually clearly separated and often even implemented in different languages. In this chapter we will talk about two programs both written in Java. The first is a user interface for GF (section 1). The second is Gramlets (section 2), where the core-interface division is more subtle.

In the subsequent sections of this chapter we will concentrate on the implementation and graphical user interface while here we want to be more theoretical in presenting the syntax editing semantics. By syntax editing semantics we mainly mean the one present in Gramlets.

The general theory behind the GF grammar formalism is Martin-Löf's Type Theory – a mathematical meta-language (or framework) for representing different logics and reasoning about them. Axioms and rules of a logic as well as functions, predicates and theorems – everything is represented as constants. Each constant has a name, a type and a definition (optional) that represent the meaning (or semantics) of the constant.

To reason within a logic, which traditionally means to prove some theorems within the logic, in type theory means to declare constants representing these theorems, i.e. to specify their name, type and definition. The type reflects the statement of a theorem while the definition is responsible for the proof. To be correct the definition (or the proof) should be of the declared type, see section 1.7 for an example.

Type theory language is expressive – by introducing new constants we can extend our logic with all the usual inductive data types and logical connectives.

The GF grammar formalism is built upon type theory language. It uses the notation with predefined keywords (such as `cat, fun`) to distinguish among the different sorts of declarations (or judgments). The main division is between the abstract (corresponds to the type of a constant in type theory) and the

concrete (has no direct analog in type theory) syntax declarations. Each abstract declaration should be completed with the corresponding concrete declaration. Abstract and concrete parts should match for the whole constant declaration to be correct.

Judgments in GF represent grammatical categories and rules. A GF grammar is a sequence of judgments and can be extended with new declarations.

ALF and Alfa proof editors allow the user to build theories by introducing new constant declarations in type theory framework. They also provide pretty-printing facilities – translating type theory into user-friendly notations. New declarations are constructed interactively by top-down step-wise refinement. One proof(definition)-constructing step corresponds to using an object formation rule in type theory. The rules are abstraction (introducing a variable, which corresponds to making an assumption) and application (using the constants already defined in the theory).

Each step is invoked in the editor by an elementary editing command, which is immediately checked by the type checker – the main part of the framework. If the step is correct, the corresponding changes are shown on the screen, otherwise an error message is generated. In this way the framework ensures that the tree built is type correct (or well-formed), i.e. corresponds to the declared type. To perform the type checking the framework must consult both the type of the declaration to be defined and the so called *environment* – constant definitions already declared in the current theory. It actually does even more, namely, analyzes the environment and the type sought and then suggests the next step, by listing the pre-approved alternatives. In case there is only one possible alternative it can even fill in the next step automatically. In non-trivial cases the next step is chosen by the user and the framework helps by narrowing down the possible choices. Thus, the framework is assisting to construct the desired definition interactively correcting and consulting the user at every step.

A similar stepwise interactive procedure is used for editing a multilingual text in GF. Here, new declarations are simply phrases we want to construct, which have certain types (such as a sentence or a verb phrase). The environment consists of the rules (the constants defined already) in a grammar (the current theory). The type checking and the corresponding next step suggestion list are also present in the GF syntax editor. Unlike proof editors, in GF the bottom-up construction is also possible.

Another useful concept borrowed from proof editors is metavariable – a placeholder for incomplete constant definition $?_i : T$. It has an expected type T and an identifier $?_i$. An identifier consists of a question mark indicating that the declaration is incomplete, i.e. intended to be replaced by a complete object; and a number assigned to tell apart different metavariables. All metavariables carry unique identifiers (numbers).

Metavariables represent the parts of the definition that are not yet refined. Therefore, a definition-constructing step is basically a metavariable-refinement

step of finding an appropriate instantiation for the metavariable.

Implementation of a type checker is the core of a logical framework. The difficulty of a type checking algorithm depends on the expressiveness of the framework.

When writing a Java GUI Syntax Editor we did not deal with the type checking problem, since all the computations were performed on the Haskell side and the Java side just displays the result for the user. The type checking algorithm for the main GF system is outside the scope of this document [7].

In case of Gramlets, which is implemented purely in Java, type checking for syntax editing is needed. However, a grammar comes to Gramlets compiled into the canonical form – a simplified (computation-oriented) representation of the grammar. Further in this section we will speak about syntax editing implemented in Gramlets although most of it applies to the GF syntax editing in general, since Gramlets' functionality is borrowed from the GF Syntax Editor.

The main functionality of Gramlets is syntax editing operations on abstract syntax tree object. In GF the user is only allowed to edit one object at a time. It can of course be saved for future references in a file, but otherwise it is not possible to have several objects during the editing session.

Syntax editing of an abstract syntax tree starts from creating a new syntax tree object of a certain type. This corresponds to the type declaration during constant declaration in type theory. Unlike for example Alfa implementation of the type theory, in GF the user is only allowed to choose among the predefined (in a grammar) types. No new types can be added on the fly. Thus, the first syntax editing step is always type declaration of the object to be built. The predefined types are always primitive in the sense that all the constrains involved are of the form $?_i{:}T$ (if no dependent type are introduced, see 1.7). Therefore, no further constraint unification is needed for type checking.

Once the type of the tree is chosen, a new syntax tree object is created. It only contains a root node, which contains a metavariable to be filled later.

Abstract syntax tree is a data structure, which has a root-node and a focus-node, where focus node works like a cursor in a text editor and points out where the current editing takes place. The tree nodes and the focus are parameters that represent so called editing *state*. In the full GF the state also includes the current grammar imported (environment). However, since the grammar is hard-wired in Gramlets the environment remains unchanged during syntax editing.

A node contains the node information and the pointers to its children- and parent- nodes. A node information can be either a metavariable (a type declaration) or a function name (constant application), whose arguments are put in the children-nodes. In figure 3 the abstract syntax tree for the expression $0{+}?_1$ is shown. Storing the type information in a node is actually needed only for metavariables, since for applied functions (predefined constants) the type information can always be looked up in the grammar (type declaration of the constant). However, we keep the type information even for function-nodes to

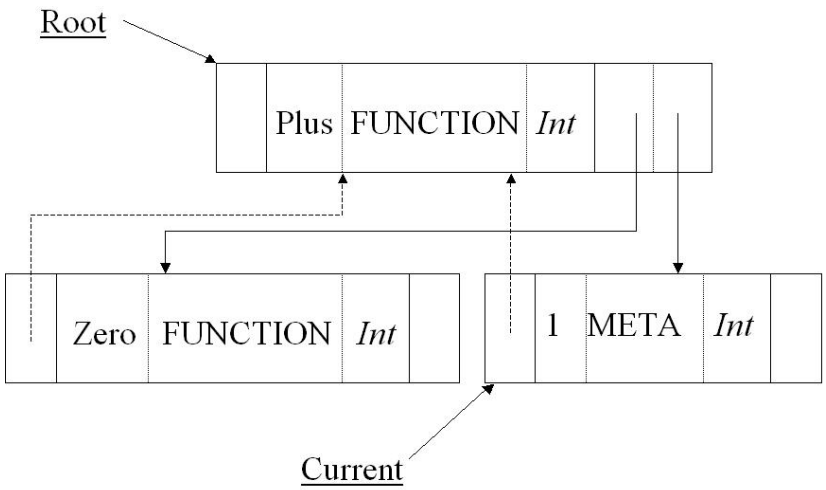

Figure 3: The abstract syntax tree for the expression $0+?_1$. The root contains the operation function `Plus`, which takes two arguments (children nodes) and returns the result if the type *Int*. One argument is the function `Zero` (with *Int* result), while the other is not yet known. Therefore, it contains a metavariable with identifier $?_1$ of the type *Int*(Integer). The field with values FUNCTION and META is used to distinct complete and incomplete nodes.

improve the performance.

Syntax editing process is replacing (or refining or instantiating) the metavariables with functions. A syntax tree is completed when there are no metavariables left. The refinement steps are suggested by the system just as during the proof construction in a proof editor. To do so the system does a simple type checking consulting the grammar (the environment) and the focus-node information (the type declaration).

Traditionally type checking is a program that type annotates abstract syntax trees parsed from a text input. However, since Gramlets do not deal with parsing, but only with text generation, the syntax trees are built already containing the type information from the start. The type-checking operations are, therefore, localized in the syntax editing commands by which the trees are built. Each refinement operation is responsible for the type checking necessary for performing the corresponding operation so that the syntax tree remains well-formed after the operation is carried out. Such localization is possible, since all the type information we need to perform a check is localized in the focus-node, not spread out over the whole tree, so it is sufficient to perform a local type check.

There are five basic syntax editing commands in GF:

- The top-down command `refine` is a standard function (constant) application in proof editors. Here we have to make sure that the type of the focus-node is the return type of the function. Children nodes containing

the metavariables of the function argument types will be introduced with this refinement operation if there are any arguments to that function. For example in Fig. 4 the focus node of the tree on the top of the picture has type `Exp`. We can use function $+ : Exp \rightarrow Exp \rightarrow Exp$ to refine the focus node, since the return type of the + is `Exp`. Two new metavariables are introduced in the resulting tree. They both have the type `Exp`, since the +-function takes two arguments of this type.

- The bottom-up command `wrap` is used on non-metavariable root- node to wrap the current tree into a bigger tree. Type checking here is simply finding the functions (constants), which have arguments of the type of the focus-node (A): $f : ...A \rightarrow ...$, if the focus node is a root or $f : ...A... \rightarrow A$, otherwise. If there is more than one argument of this type several alternatives will be presented, one for each occurrence. After wrapping operation the function arguments different from the focus-subtree if any will be represented by the metavariables of the appropriate types. For example in Fig. 4 we can wrap the focus node of the middle tree with function $succ : Exp \rightarrow Exp$. The resulting tree will comprise the + subtree with the *succ* node on the top. No extra metavariables are introduced, since *succ* takes only one argument of the type Exp.

- `ChangeHead` is performed on nodes that contain a function that can be replaced with another function of the same type while keeping the old argument subtrees. Type checking in this case is comparison of the old and the new function types. Corresponding argument types as well as the return type should be the same for both functions. No new metavariable nodes will be introduced with this operation. For example in Fig. 4 function $+ : Exp \rightarrow Exp \rightarrow Exp$ can be replaced by $* : Exp \rightarrow Exp \rightarrow Exp$ function with the same type signature. Arguments stay the same while the top node + is replaced by $*$. The resulting tree is shown in the right down corner.

- `PeelHead` is used if the focus node contains a function of the type: $f : ...A \rightarrow ...$, if the focus node is a root or $f : ...A... \rightarrow A$, otherwise; to remove the function while keeping (some of) the argument subtrees. Can be seen as opposite to the wrap command. To perform the peel operation correctly we need to check that the function actually has such a type. No new metavariable nodes will be introduced with this operation. For example in Fig. 4 the *succ* head can be peeled off. This transforms the tree in the left down corner back into the middle tree.

- `Delete` (local undo) operation replaces the subtree in focus with a metavariable of the proper type. To specify the correct type we need to look up the type of the function (constant) specified in the focus node. For example in

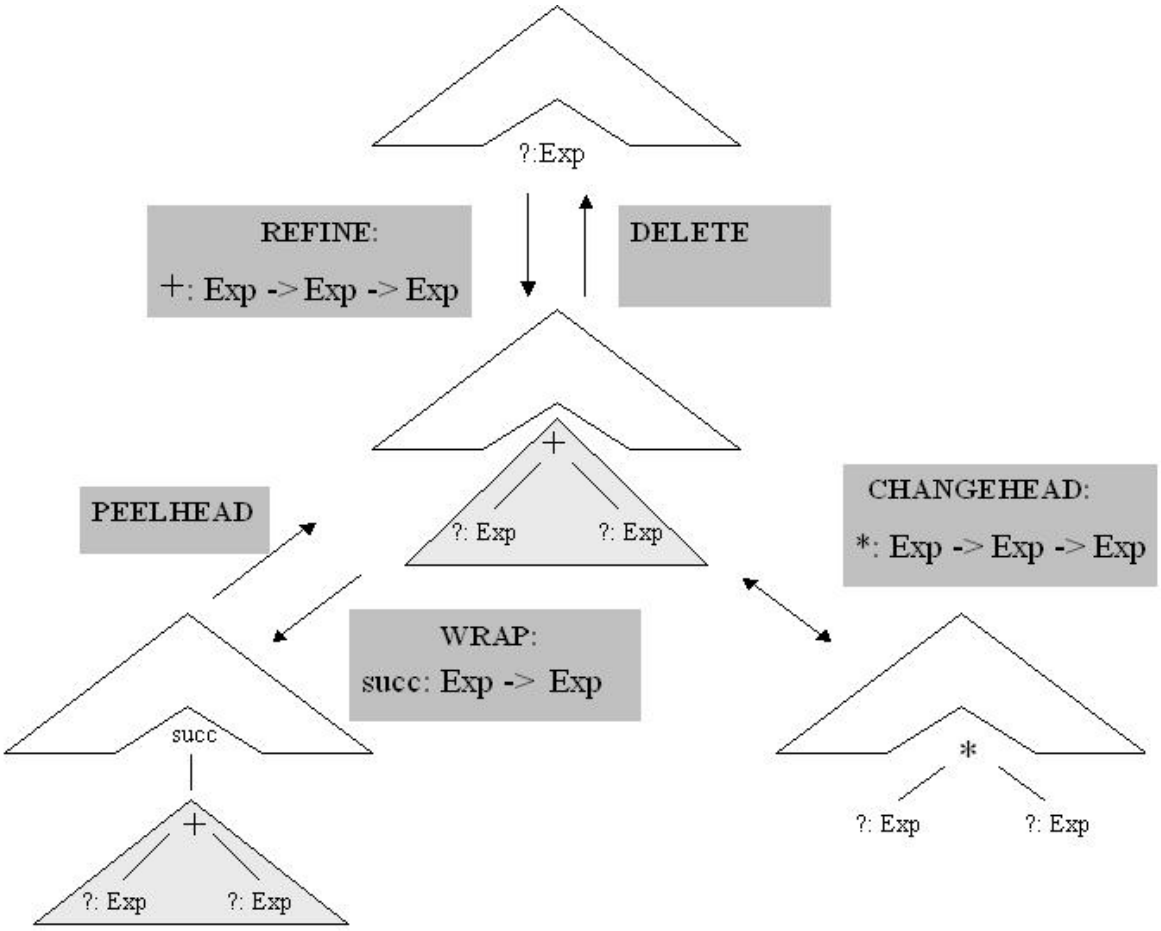

Figure 4: Examles of 5 basic types of syntax editing commands in GF. Syntax trees contain hidden parts (shown as roofs) and the focus nodes with its subtrees (if any). The original abstract syntax tree is the top tree. Arrows show possible refinement steps. The type of the editing step and the type signature of the refinement function are shown near the corresponding arrow.

Fig. 4 delete operation performed on the middle tree will lead back to the tree on the top, so the whole +-subtree will be removed and replaced by a metavariable of the type *Exp*.

The Gramlets syntax editor only proposes (by showing in the editing menu) the editing steps that are pre-approved during type checking. The type checking procedure consists of simply looking at the focus-node type information and then fetching the appropriate functions (functions with the return type of the focus-node) from the grammar. Delete and refine operations are standard for proof editors. For syntax editing examples using the Java GUI editor see section 1.

As we have mentioned Gramlets are only able to work in the direction of text-generation, not parsing. The process of translating an abstract syntax tree into a text is called linearization. This is done by single-pass traversal of the syntax tree by using the canonical GF computational model. Linearization transforms an abstract syntax tree into sequences of terminals by using linear patterns defined in the linearization rules (constant definitions). The linearization rules are required to be compositional i.e. the linearization of a constant is a function of the linearizations of its arguments. This allows us to treat argument variables as pointers to the linearizations of subtrees, which in turn makes the linearization algorithm efficient enough to produce linearizations of trees on the fly. The linearization of an incomplete abstract syntax tree containing metavariables is shown to the user as a feedback during the syntax editing of a tree. Update in a tree invokes the corresponding update in the linearization. Producing a completed (without metavariables) text from the abstract syntax tree corresponding to the meaning of the text is the goal of the syntax editing in GF.

The canonical representation of a grammar is essentially a table where we can easily look-up the type information as well as linearization patterns (constant definition) needed for type checking and linearization.

Two supporting syntax editing operations *undo* (chronological) and *random* are also implemented in Gramlets. For *undo* we just keep the record of the trees at the last editing steps. Random generator makes the choice of the next step among the pre-approved alternatives until all metavariables are eliminated from the tree. The random operation is convenient, for example, for the demonstration and surface testing of a grammar.

Two subsequent sections are about two different syntax editing programs: Java GUI Syntax Editor (GUI modules in Java) and Gramlets (standalone Java program). The next section describes the Java GUI Syntax Editor. Syntax editing operations in Gramlets is a subset of those in Java GUI Syntax Editor, so to avoid repetition we will just talk about the Gramlets implementation and general motivation behind the project.

1 Java GUI syntax editor for GF

Grammatical Framework (GF) forms a basis on which various Natural Language Processing (NLP) applications can be built. Java Syntax Editor provides a Graphical User Interface (GUI) for GF. Together with the editor the GF system can be used as a multilingual document authoring tool. The Java GUI Syntax Editor is intended for work on the author level. This section describes the Java Syntax Editor program and presents a simple example of the GF syntax editing session. The content of this section overlaps with [20, 18, 16].

The main purpose of the Syntax Editor is to construct a text simultaneously in several natural languages. The author does not have to know all the languages represented, but the GF system assures that if the output is correct in at least one of them, including the GF abstract language - language-independent semantic representation, then it will be syntactically and semantically correct in the rest of the languages. This reflects the idea of so-called multilingual authoring.

The core of the GF system is written in a functional programming language Haskell. Actually, there is a number of user interfaces available for the GF: command line mode, ALFA proof editor, Fudget Syntax Editor and Java Syntax Editor. The subject of the paper is the latter and the latest one - Graphical User Interface (GUI) written in an imperative programming language Java. All the rest belong to functional programming. Java was chosen as an implementation language for GF user interface due to the following main reasons:

- Cross-platform
- Unicode support
- Extensive GUI libraries

The GF - GUI architecture takes a standard client-server approach (Fig. 5). The GF executable (Haskell source)plays the server role. Java GUI classes interpreted by Java Virtual Machine (JVM) form a client. The communication protocol consists of GF command string sent by the client, which uses the standard controls like buttons and menus in order to issue corresponding request, and GF result string in XML-format. GF commands used are roughly the same as in the command line mode. The GF result string is processed on the client side to be fitted into the GUI controls.

1.1 Editor's structure

We will describe the functionality of the Java GUI Syntax Editor by examples. Before we start with the examples let us look at the general appearance of the syntax editor in Fig. 6.

The main areas are:

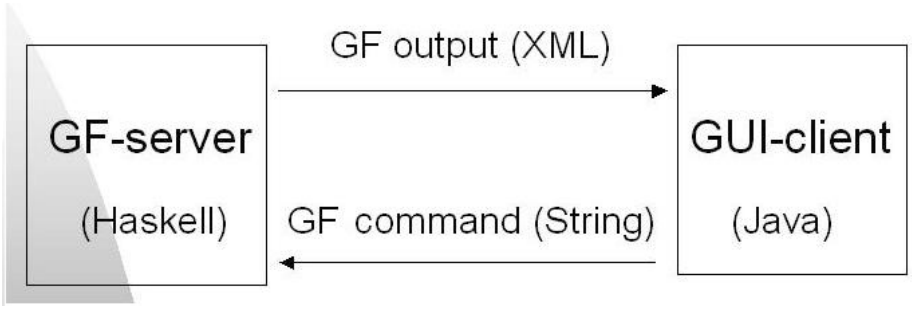

Figure 5: The communication between GF and Java GUI is performed according to the client-server architecture.

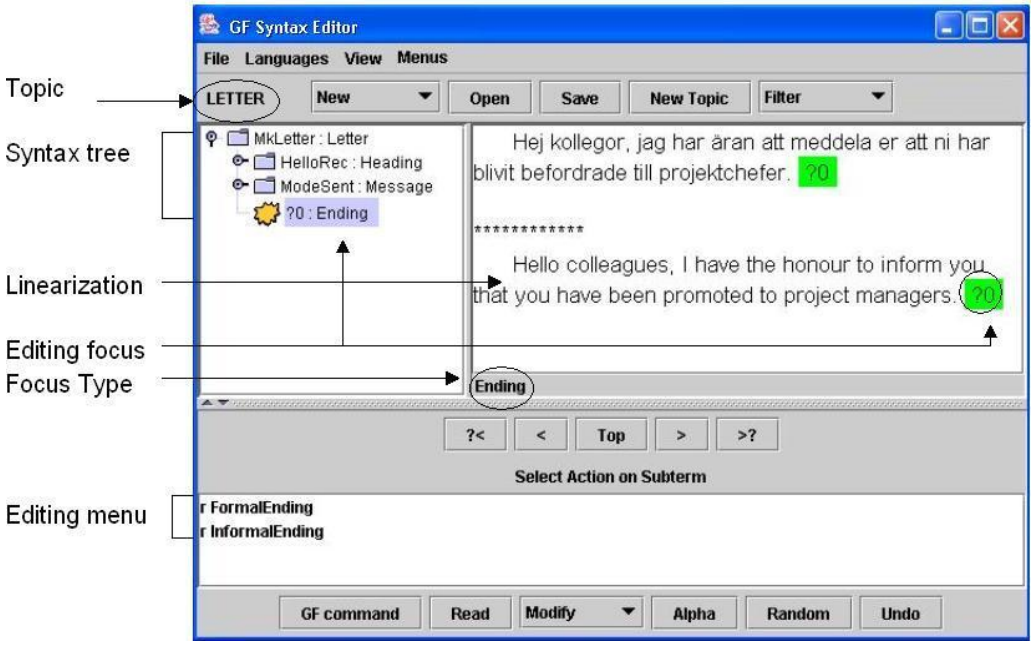

Figure 6: . Java GUI Syntax Editor's structure.

- *Tree Panel* – displays the abstract syntax tree (AST) representation of the edited object.
- *Linearizations Panel* – shows the linearizations corresponding to the AST in different languages. In Fig. 6 there are linearizations in Swedish and English. The linearization panel is an editable area. One can click on the chosen word (or select a group of words) to shift the focus. Middle-click allows you to type directly in the text.
- *Editing Menu Panel* – contains the refinement options for the current focus. You can also get the refinement list in a pop-up menu invoked by a mouse right-click on the chosen tree node of directly in the text.

Other common elements are:

- *Topic* – says to what domain the current editing object belongs. The topic is extracted from the grammar file name. In Fig. 6 the topic is LETTER, which means that the user is building a letter according to the GF letter grammar.
- *Focus* – Colors background selection marks the editing focus. Focus is highlighted both in the tree and the linearizations, since they are just parallel representations of the same editing object. One can change the focus by clicking on a new tree node or in the text. One can also "select" several words to get the focus that spreads across more than one word.
- *Focus Type* – specifies the syntax type of the editing focus. In Fig. 6 the focus type is Ending , which means that the user is now constructing the final piece of the letter.

For a more systematic explanation of GUI functionality take a look at appendix 1.8.

1.2 Creating a new object

When you start the GF editor the topic and the languages you want to work with should be chosen. Let us say, we want the LETTER topic in four languages: English, Swedish, French and Finnish. The topic can be changed later at any moment. You can create a new editing object by choosing a category from the *New* list. For example, to construct a letter, choose the *Letter* category (Fig. 7). In Fig. 8 you can see the created object in the tree form in the left upper part as well as linearizations in the right upper part. The tree representation corresponds to the GF language-independent semantic representation, the GF abstract syntax or interlingua. The linearizations area displays the result of translation of abstract syntax representation into the corresponding language using the GF concrete syntax.

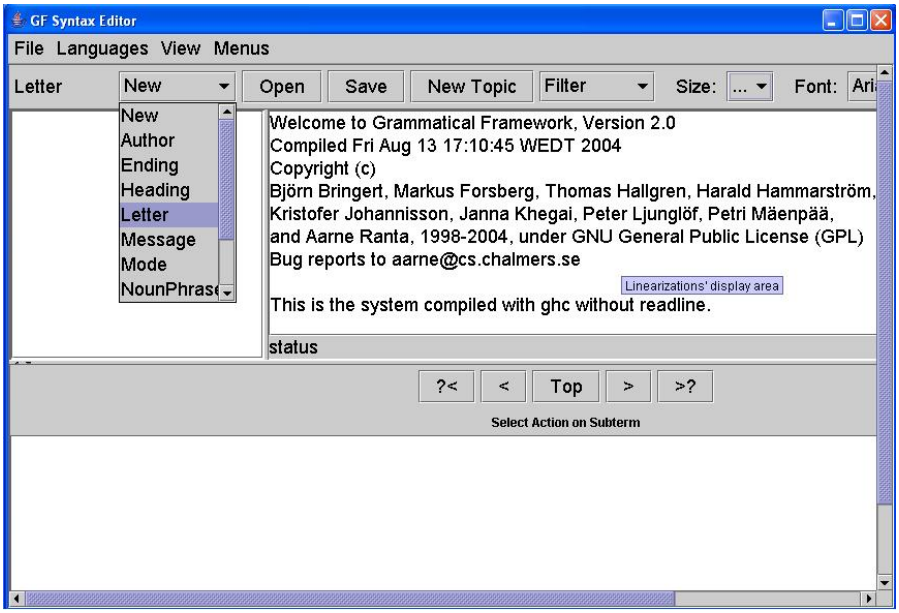

Figure 7: The New menu shows the list of available categories within the current topic LETTER. Choosing the category Letter in the list will create an object of the corresponding type. The linearizations area contains a welcome message when the GF Editor has just been started.

1.3 Refining the object

According to the LETTER grammar a letter consists of a Heading, a Message and an Ending, which is reflected in the tree and linearizations structures. However, the exact contents of each of these parts are not yet known. Thus, we can only see question marks, representing metavariables, instead of language phrases in the linearizations.

Editing is a process of step-wise refinement, i.e. replacement of metavariables with language constructions. In order to proceed you can choose among the options shown in the refinement list. The refinement list is context-dependent, i.e. it refers to the currently selected focus. For example, if the focus is Heading, then we can choose among four options. Let us start our letter with the DearRec structure (Fig. 9(a)).

In order to see the linearizations in three languages at the same time we have to first choose the *text* mode in the *Filter* menu, see the upper panel buttons description. This will make the letter look more compact without extra free lines between the letter parts. Second, we have to scroll down. Alternatively to the second step one can switch off the abstract representation: see subsection 1.4.

Now we have a new focus - metavariable ?4 of the type Recipient and a new set of refinement options. We have to decide what kind of recipient the letter has. Notice that the word *Dear* in Swedish and French versions is by default in male

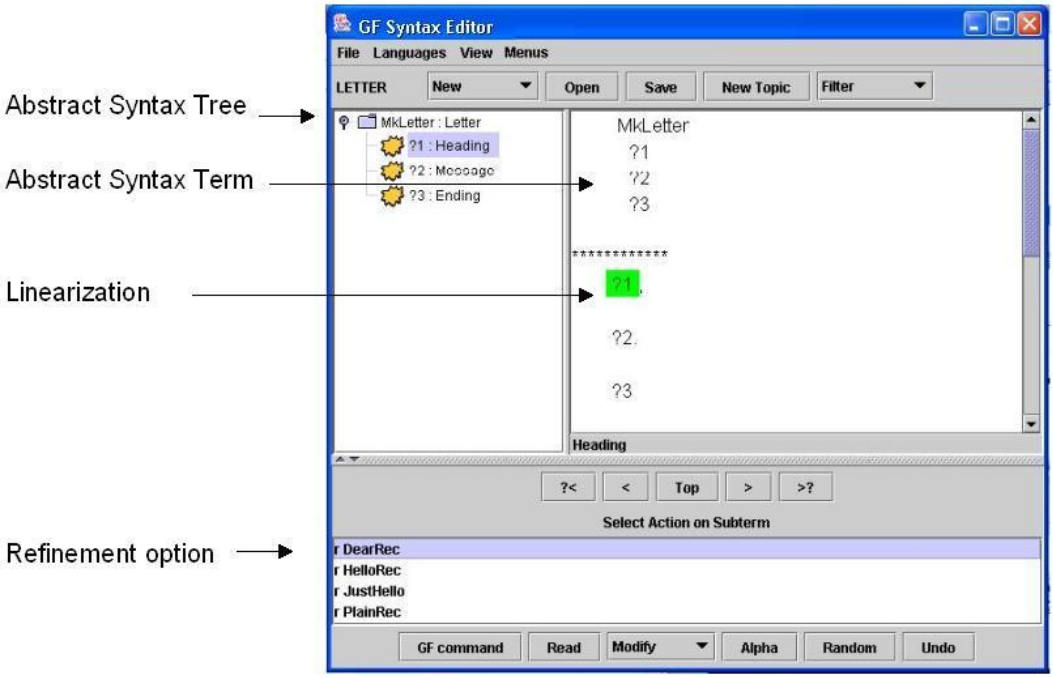

Figure 8: The Abstract Syntax tree represents the letter structure. The current editing focus, the metavariable ?1 is highlighted. The type of the current focus is shown below the linearizations area. The context-dependent refinement option list is shown in the bottom part.

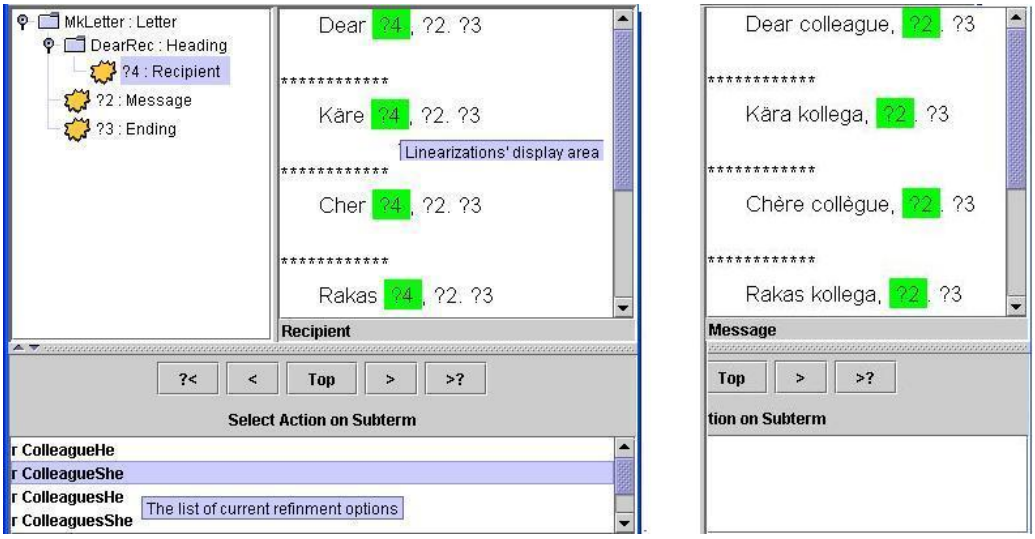

Figure 9: (a) The linearizations are now filled with the first word that corresponds to *Dear* expression in English, Swedish, French and Finnish. The refinement focus is moved to the Recipient metavariable. (b) The Heading part is now complete. The adjective form changes to the corresponding gender after choosing the recipient.

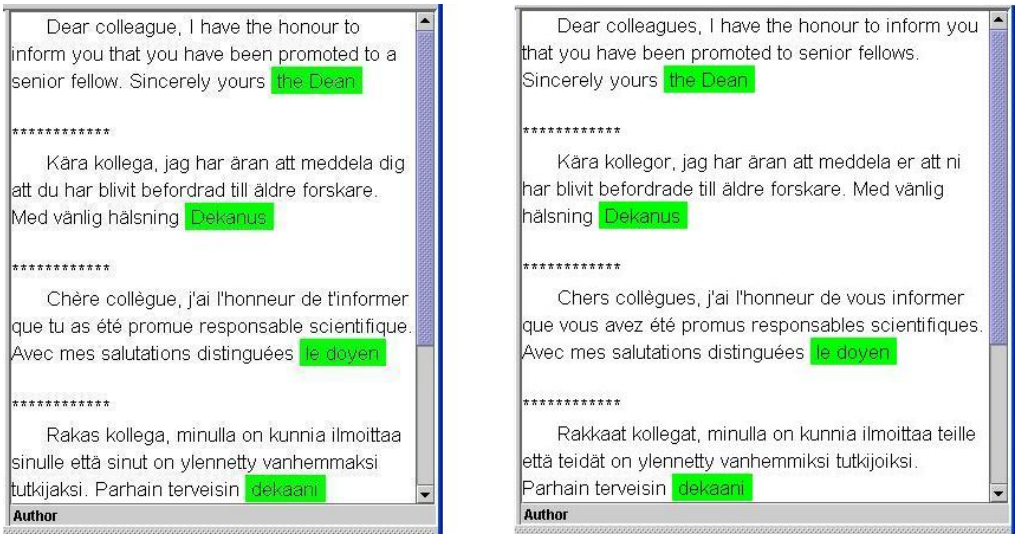

Figure 10: (a) The complete letter in four languages. (b) Choosing the plural male form of the Recipient causes various linguistic changes in the letter as compared to (a).

gender and, therefore, uses the corresponding adjective form. Suppose we want to address the letter to a female colleague. Then we choose the ColleagueShe option (Fig. 9(b)).

Notice that the Swedish and French linearizations now contain the female form of the adjective *Dear* , since we chose to write to a female recipient. This refinement step allows us to avoid the ambiguity while translating from English to, for example, a Swedish version of the letter.

Proceeding in the same fashion we eventually fill all the metavariables and get a completed letter like the one shown in Fig. 10(a).

There are six types of commands appearing in the *Select Action* window:

- *r (refine)* – used on metavariables to refine them.
- *w (wrap)* – used on non-metavariables of some type A to wrap them by functions of the form: $f : ...A \to ...$, if the focus node is a root or $f : ...A... \to A$, otherwise .
- *ch (changeHead)* – used on functions to replace the current function with another function of the same type while keeping the old argument subtrees.
- *ph (peelHead)* – used on functions of the type: $f : ...A \to ...$, if the focus node is a root or $f : ...A... \to A$, otherwise; to remove them while keeping (some of) the argument subtrees. Can be seen as opposite to the wrap command.
- *d (delete)* – used on non-metavariables to delete them.

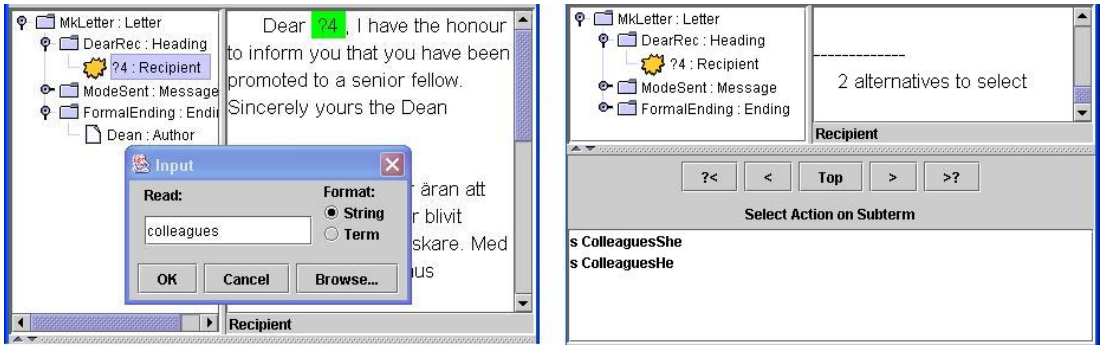

Figure 11: (a) A refinement step can be done by using the *Read* button, which asks the user for a string to parse (only in English in the present version).(b) When the parsed string in (a) is ambiguous GF presents two alternative ways to resolve the ambiguity.

- *s (select)* – used after ambiguous parsing or paraphrase.

Refinement steps can also be generated randomly, by clicking on the Random button.

A completed letter can be modified by replacing parts of it. For instance, we would like to address our letter to several male colleagues instead. We need first to move the focus to the Header node in the tree and delete the old refinement. In Fig. 11(a), we continue from this point by using the Read button, which invokes an input dialog, and expects a string to parse. Alternatively we middle-click directly in the text and a special editing text field appears. Let us type *colleagues*.

The parsed string was ambiguous, therefore, as shown in Fig. 11(b), GF asks further questions. Notice that after choosing the ColleaguesHe option, not only the word *colleague* , but the whole letter switches to the plural, male form, see Fig. 10(b). In the English version only the noun *fellow* turns into plural, while in the other languages the transformations are more dramatic. The pronoun *you* turns into plural number. The participle *promoted* changes the number in the Swedish and French versions. The latter also changes the form of the verb *have*. Both the gender and the number affect the adjective *dear* in French, but only the number changes in the corresponding Finnish adjective. Thus, the refinement step has led to substantial linguistic changes.

1.4 Adding new languages

So far all the available languages have been displayed. However, some of them can be switched off using the *Languages* menu. To add a new language, one has to work on a *concrete syntax*. Target languages can be added on the fly: if a new language is selected from the Language menu, a new view appears in the editor

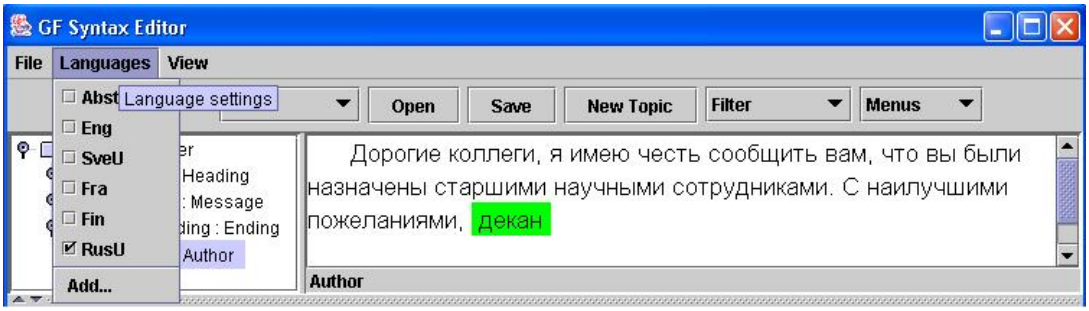

Figure 12: Now we are able to translate the letter into Russian.

while other things remain equal, including the document that is being edited. Fig. 12 shows the effect of adding Russian to the above example.

1.5 Saving the object to a file

We can save our work by clicking the *Save* button. In the open dialog we should specify the name of the file as well as navigate to the directory where the file will be placed. The default directory is the GF directory under Windows platform or the running directory otherwise. We can also choose the saving format: Text or Term. The Term option will save the abstract syntax representation, while the Text option – the linearizations displayed at the moment. Both types of documents can be later opened by the editor. However, it is safer to open terms, because texts may be impossible to parse since linearization may destroy important information. For example, is we save the English version of the letter containing *Dear colleague* - heading the gender information of the *colleague* word will be lost. In Fig. 13 we want to save the language-independent, abstract representation in the file named myLetter in the GF directory.

1.6 Changing the topic

The LETTER domain is restricted to constructing letters. Several other sample grammars are provided with GF and also the user can write his own grammars. To create a new subject matter (or modify an old one), one has to create (or edit) an *abstract syntax*. When you want to work in a different domain, assuming that the corresponding grammar (both abstract and concrete parts) is written, you can use the *New Topic* button. You will get an open dialog, where you should navigate to the grammar file and specify the file name. Windows users have the grammar files stored in the Grammars subdirectory of GF directory. In Fig. 14 we are about to download the grammar from arithmetic.Eng.gf file, where .gf is the extension of GF grammar files, Eng tells that the file describes the linearizations in English and logic is the topic name that will be shown on the upper button panel.

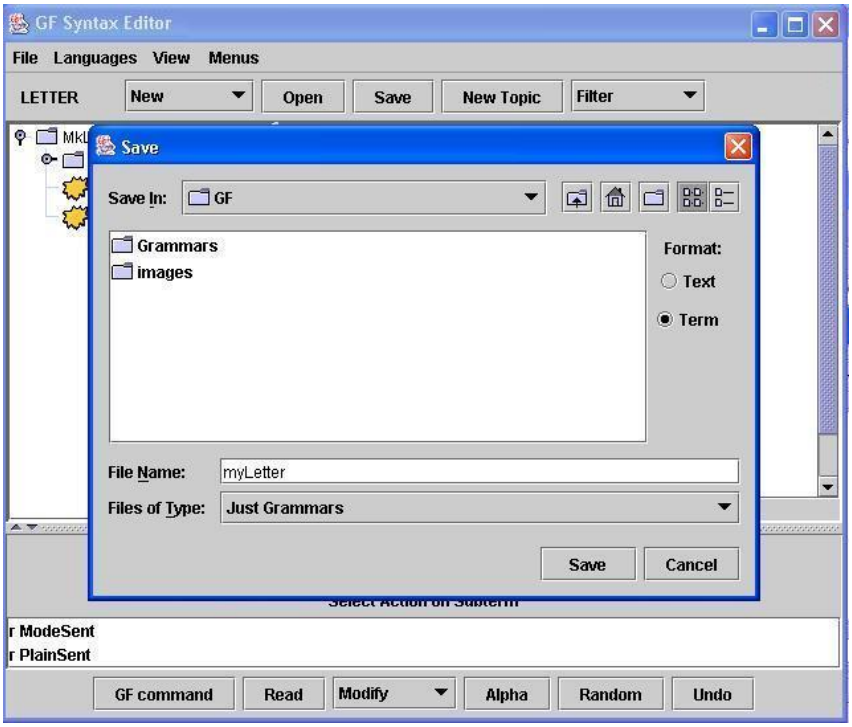

Figure 13: Pressing the Save button brings up a file chooser dialog.

After choosing the file with a new grammar you will get a picture very similar to the one in the beginning of this example, except for the different topic and, correspondingly, a new list of available categories, and a different welcome message Fig. 15.

1.7 More syntax editing commands

The ARITHMETIC grammar allows us to illustrate some commands, which were missing in the previous examples, namely, *wrap* and *peelHead* refinement commands as well as three commands from the *Modify* menu: *compute*, *paraphrase* and *solve*.

Let us start with a simple construction shown in Fig. 16. It contains a theorem and its proof, which is rather trivial, since it just refers to an axiom from the grammar.

Notice that unlike the LETTER grammar ARITHMETIC grammar can also be treated as a formal mathematical theory describing arithmetical domain. The ARITHMETIC grammar, therefore, contains definitions, axioms and deduction rules, which allows us to formulate and prove theorems about the arithmetic domain similarly to what one can do in proof editors [23, 14].

The current focus node *zero* can be wrapped, for example, with *succ* function. The result is shown in Fig. 17. So, now instead of the statement *Zero is even* we

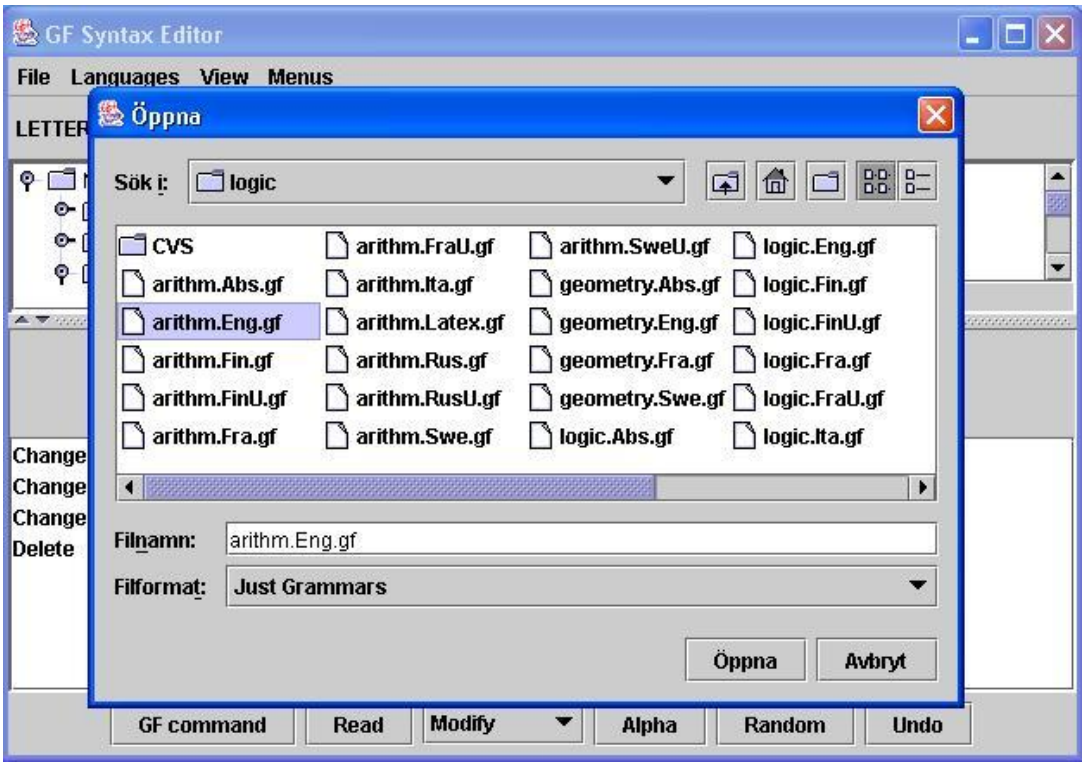

Figure 14: Pressing the *New Topic* button brings up an open dialog to choose a grammar file.

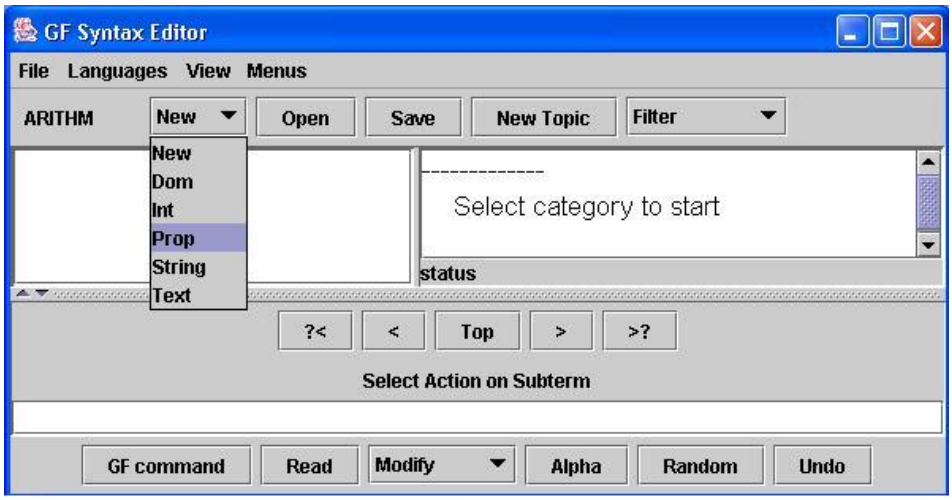

Figure 15: GF Syntax Editor after choosing the ARITHM (ARITHMETIC) grammar.

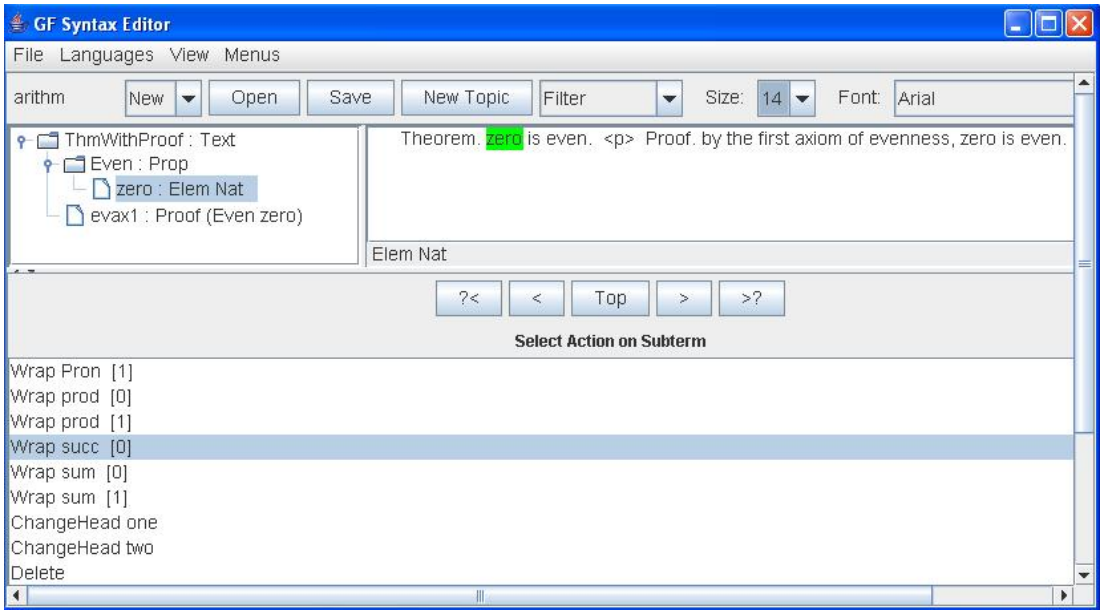

Figure 16: Zero node can be wrapped with *succ* function. This is possible, since the *succ* function takes a natural number as an argument and returns the result of the same type.

have the statement *The successor of zero is even.* Although the new statement is correct from the linguistical point of view, the mathematical proposition is wrong (and, therefore, highlighted with read color), since the type checker finds out that the proof and the proposition do not match. We changed the proposition while the proof part remains the same. However, in the ARITHMETIC grammar, as in mathematics in general a proof and a proposition in a theorem are not irrelevant to each other. Namely, the theorem proof should prove the proposition of the theorem. For this purpose, the *dependent types* are used. Thus, the type of proof is dependent on the proposition type. This leads to type-checking procedure invocation each time we change some part of the theorem. Type-checking makes sure that the proof and the proposition still conform to each other. The result of the type-checking is a number of *constraints* that should be met in order to keep the theorem correct. In Fig. 17 we can see such a constraint in the root node of the tree in brackets: *zero <>succ zero*. This simply says that the theorem is correct as soon as zero is equal to the successor of zero.

To go back to the correct proposition we can use the *peelHead* refinement option, which will basically undo the effect of the wrap operation. However, this is true only for a wrap function, where both the result and one of the arguments are of the same type. Otherwise, we could not peel and would have to use the *Undo* button to restore the original proposition.

Another operation that can be demonstrated here is *paraphrase* from the *Modify* menu, see Fig. 17. As the result of the paraphrase search we get two

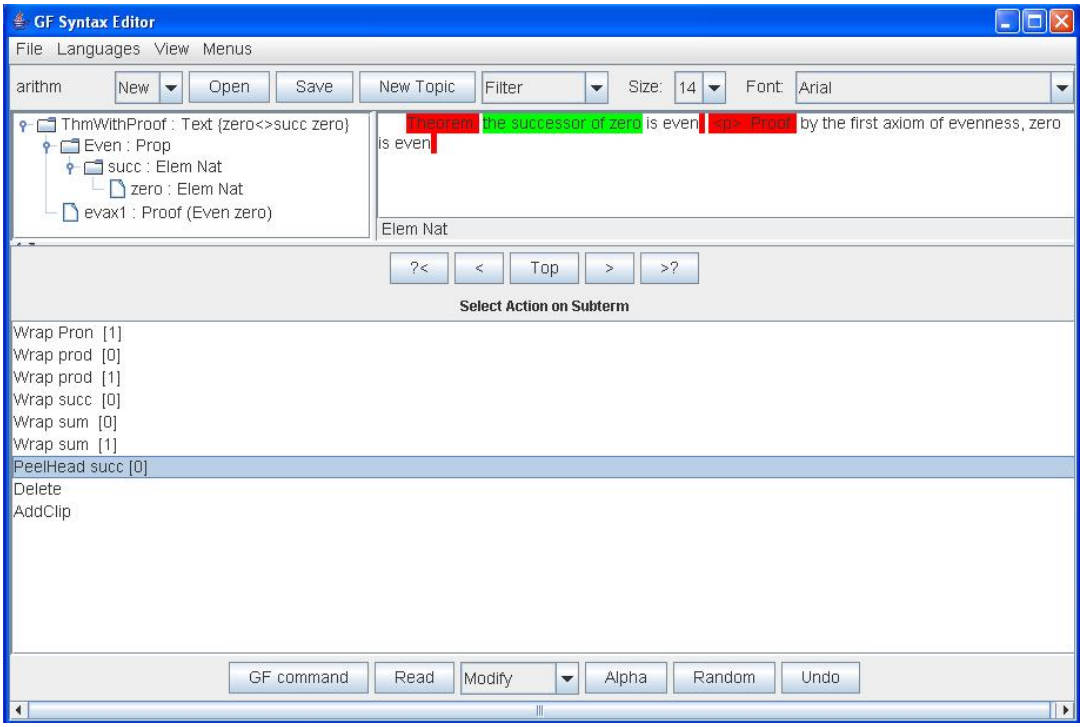

Figure 17: Constraint *zero <>succ zero* is produced by the type-checking procedure to assure that the theorem type is correct, namely, the proof part contains the proof of the stated proposition. *PeelHead* selection option undo the wrapping operation and restore the proposition in Fig. 16. This is only possible for functions that have at least one argument of the same type as the result type. *Paraphrase* operation allows us to search for equivalent expressions.

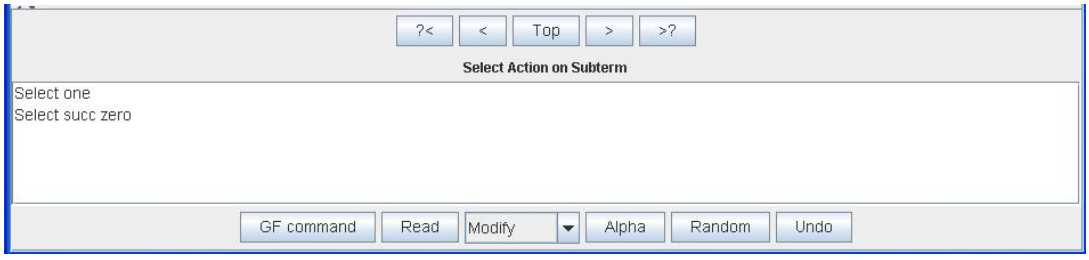

Figure 18: There are two versions (synonyms): *one* and *the successor of zero* to choose from.

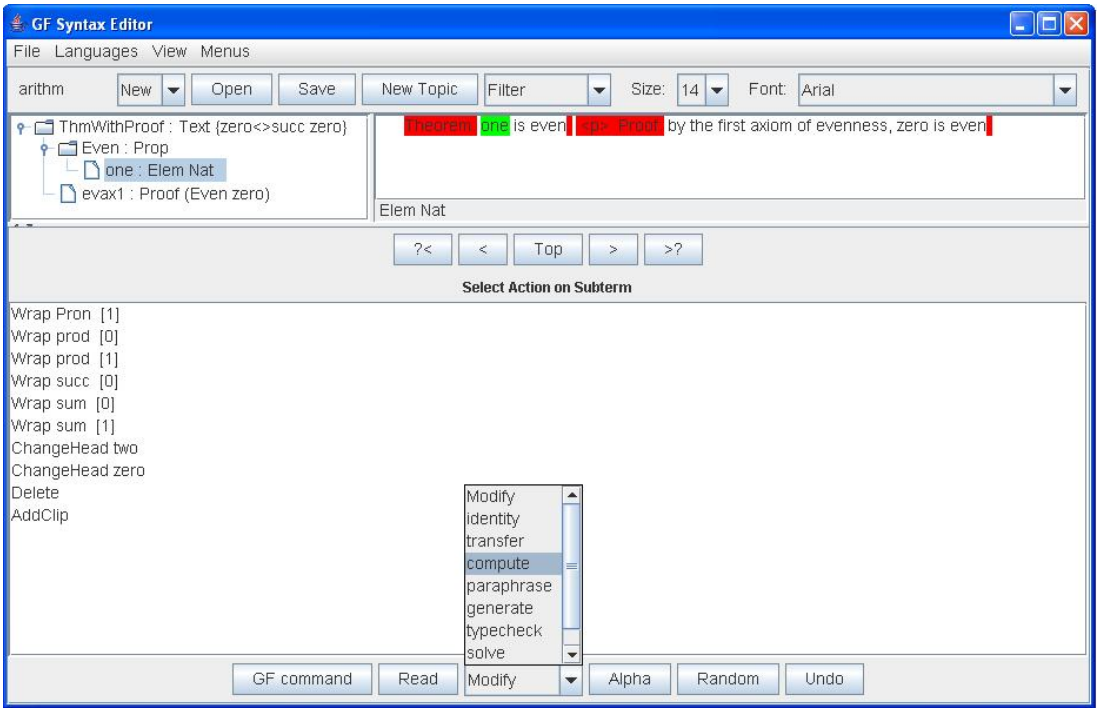

Figure 19: Compute operation will unfold the definition of *one* giving *the successor of zero*.

selection options, see Fig. 18. Namely, we can keep *the successor of zero* or use stylistically nicer *one* instead, see Fig. 19. Of course, such variations are possible due to the corresponding definitions made in the ARITHMETIC grammar. For instance, the function *one* is defined as the successor of zero *succ zero*. If we want to go back to the longer successor expression we can use the operation *compute* from the *Modify* menu, see Fig. 19. It unfolds the function definition if any and computes the linearization of the result.

Dependent types allow us to even fill parts of a theorem by using the *solve* operation from the *Modify* menu, see Fig. 20. Here, we do not have an argument for the *even* function, although we have a completed proof, which is dependent on the proposition. *Solve* operation resolves the constraint *zero <>?0* necessary for the theorem being correct. In this case it is only possible if the argument to the *even* function is *zero*, which gets us back to Fig. 16.

More systematic description of the Java GUI Syntax Editor controls can be found in the Appendix I. The GF commands that are not accessible via GUI controls can still be sent to GF as a command line using the *GF command* button.

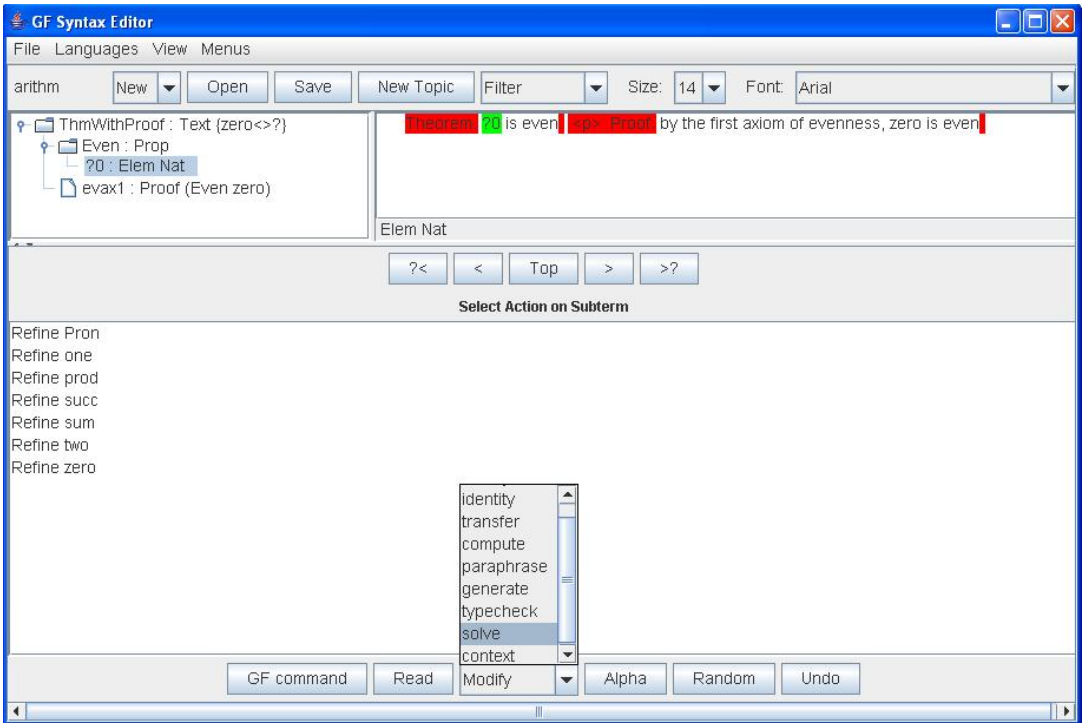

Figure 20: *Solve* operation will resolve the type constraint *zero <>?0* (in the root node) and fill the metavariable *?0*.

1.8 Java GUI Editor command reference

Here we describe the functionality of the editor GUI controls. There are two main possibilities to access the Editor functions: the menu bar and the button panels. Some operations are accessible both from the menu and the buttons. Menu items with three dots like in *Open...* assume that a file-chooser will appear before performing reading or writing operation.

File menu

File menu contains the main operations:

- *Open...* – Read both a new environment and an editing object from a file. The current editing will be discarded, but first the user will be prompted to confirm his/her intentions.
- *New Topic...* – Read a new environment from a file. The old work will be dismissed although a warning message will be displayed beforehand.
- *Reset* – Empty the environment. The objects created previously will be lost. The user will always be asked for permission to perform the operation.
- *Save As...* – Write the current editing object to a file in the term or text format.
- *Exit* – Quit the editor.

Languages menu

Languages menu Controls the language settings. First, it contains the list of available languages including so-called Abstract, language-independent syntax representation Abs. Only languages with marked checkboxes will be shown in the linearizations' display area.

- *Add...* – Add another language to the current topic by reading the corresponding grammar file.

View menu

View menu controls the appearance of the editor:

- *Tree* – Show/hide the tree representation of the current editing object.
- *One window* – Put all the panels in one window.
- *Split windows* – Put the editing selection menu in a separate window.

Menus menu

Change the display format of the refinement options:

- *language* – show the menu through linearization in the corresponding language or as formal object (default).
- *short* – use short command names (default).
- *long* – use long command names.
- *typed* – show types of refinements.
- *untyped* – don't show types of refinements (default).

Upper panel buttons

The operations affecting the environment state:

- *New* – Start a new goal of the chosen type. The current editing will be discarded, but first the user will be promted to confirm his/her intentions.
- *Open* – Read both a new environment and an editing object from file. The old work will be dismissed although a warning message will be displayed beforehand. Duplicates the Open... item in the File menu.
- *Save* – Write the current editing object to a file in the term or text format. Duplicates the Save... item in the File menu.
- *New Topic* – Read a new environment from file. The objects created previously will be lost. The user will always be asked for permission to perform the operation. Duplicates the New Topic... item in the File menu.
- *Filter* – Apply the chosen Filter (one of the -filter values) to the linearization output:
  - *identity* – No change (default).
  - *erase* – Erase the text.
  - *take100* – Show the first 100 characters.
  - *text* – Format as text (punctuation, capitalization).
  - *code* – Format as code (spacing, indentation).
  - *latexfile* – Embed in a LaTeX file.
  - *structured* – show with constituents in brackets.
  - *unstructured* – don't show brackets (default).
- *Size* – Choose the font size.
- *Font* – Choose the font for linearization display area.

Middle panel buttons

Here, the buttons related to the tree navigation are collected:

- *?<* – Go to the previous metavariable.
- *<* – Go one step back (up)in the tree.
- *Top* – Go to the top of the tree.
- *>* – Go one step ahead in the tree.
- *>?* – Go to the next metavariable.

Bottom panel buttons

This panel contains the refinement-related operations:

- *GF command* – Send a string command to GF. The button is meant for advanced users. For GF command syntax see [32].
- *Read* – Read a term or parse a String as a refinement of the current sub goal. The input can be either typed or read from a file.
- *Modify* – Transform the current term:
  - *identity* – don't change (default).
  - *compute* – compute to normal form.
  - *paraphrase* – generate trees with the same normal form.
  - *typecheck* – perform global typecheck.
  - *solve* – apply global constraint solver.
  - *context*– try to refine with variables bound in context.

  The compute and paraphrase commands only have effect if the grammar has semantic definitions (def judgements). The typecheck and solve commands only have effect if the grammar has dependent types. The context command only has effect if the grammar has variable bindings.
- *Alpha* – Change (alpha convert) a bound variable. The syntax is: "x_0 y" means to change x_0 to y. This command only has effect if the grammar has variable bindings, and if the current focus has variable bindings.
- *Random* – Find a random refinement.
- *Undo* – Go back in the refinement history.

2 Gramlets: GF on-line and in the pocket

Gramlet is a Java applet/application with syntax editing functionality (described in section 1) for a specialized grammar [17]. It is another user-related branch in GF development, whose name is a combination of the words *GRAMmatical framework (GF)* and *appLETS*. The main purpose is to make GF more accessible for wider audience. This is to be achieved by better portability and easier installation and usage.

The Gramlets project is the first attempt to implement the GF functionality purely in an imperative programming language, namely, Java known for its portability. This makes Gramlets more portable, since they do not need the platform-dependent executable (written in Haskell) necessary for the main GF system.

Gramlets written in Java are aimed to work on PDA (Personal Digital Assistant) devices that support Java. Our target PDA is Sharp Zaurus SL-5500 handheld computer [40], which has a JVM (Java Virtual Machine) compatible with the platform. To easily install and run a Gramlet on a PDA a special installation package is prepared.

A gramlet as a Java applet can be run in an internet browser (provided that the corresponding Java Plug-ins are downloaded). Therefore, running Gramlets is fast and easy. Gramlet example in the MS Internet Explorer is shown in Fig. 21 For on-line example visit the Gramlets homepage [12].

A light-weight, portable Java applet of course does not possess the full GF functionality. It is simply a syntax editor, that can be used as a multilingual authoring tool for a predefined topic. For example in Fig. 21 we can see the Health gramlet where the user can construct some statements about somebody's health condition. Unlike the normal Syntax Editor, where a new grammar can be loaded, the grammar in a Gramlet is hard-wired. In case one wants to work with a different topic one needs to produce another Gramlet specialized for that particular grammar. Fortunately, producing a new gramlet is an automatic process. A command script can be used to generate a gramlet given a GF grammar, GF binary (compiled Haskell) and a number of Java classes from the Gramlets project. Therefore, one does not need to do any extra programming oneself, just specify the input grammar and execute the script. One can also produce a gramlet for a grammar one wrote oneself and run it on PDA or put it on WWW. However, the grammar used for Gramlets production should not contain dependent types, since the Gramlets implementation does not have the full strength of the GF grammar formalism.

2.1 Canonical GF

The full GF grammar formalism allows the use of *function definitions* and *pattern matching* mechanism that raise the level of abstraction of the grammarian work.

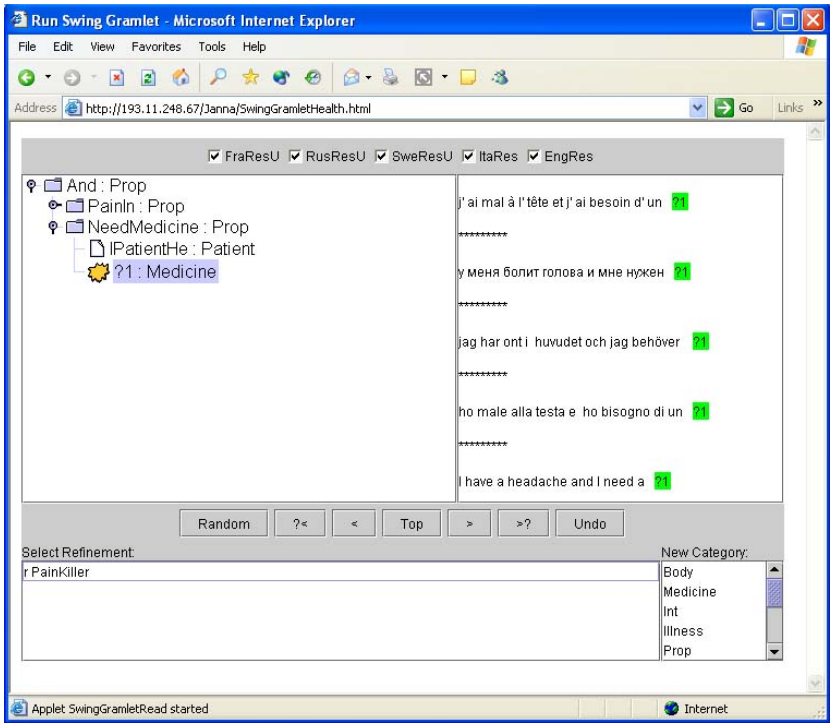

Figure 21: Health gramlet in the Internet Explorer browser.

However, for the simpler computation the grammars in this notation must be normalized into so called canonical GF by type driven partial evaluation[30]. This format of the GF grammar is produced on GF side, which makes further processing on the Java side simpler.

The type driven partial evaluation originates as GF itself from the functional programming and corresponds to program evaluation if the GF grammars are regarded as functional programs (which they in fact are). In [17] the following example of partial evaluation is given. Consider the rule `DefOneNP`:

```
fun
  DefOneNP: CommNounPhrase -> NounPhrase ;         -- "the car"
lin
  DefOneNP = defNounPhrase Sg ;
oper
  defNounPhrase : Number -> CommNounPhrase -> NounPhrase =
    \n,car ->
  {s = \\c => artDef ++ car.s ! n ! toCase c ; n = n ; p = P3} ;
```

where the following is predefined:

```
param
  Number = Sg | Pl ;
```

```
  Gender = NoHum | Hum ;
  Case   = Nom | Gen ;
  Person = P1 | P2 | P3 ;
  NPForm = NomP | AccP | GenP | GenSP ;
oper
  artDef = "the" ;
  toCase  : NPForm -> Case = \c -> case c of
      {GenP => Gen ; _ => Nom} ;

  CommonNounPhrase: Type = {s : Number => Case => Str; g : Gender} ;
  NounPhrase: Type = {s : NPForm => Str ; n : Number ; p : Person} ;
```

The canonical GF representation of the same function after partial evaluation will be:

```
lin DefOneNP = \CN_0 -> { s = table {
  NomP  => "the" ++ (CN_0.s ! Sg) ! Nom ;
  AccP  => "the" ++ (CN_0.s ! Sg) ! Nom
  GenP  => "the" ++ (CN_0.s ! Sg) ! Gen
  GenSP => "the" ++ (CN_0.s ! Sg) ! Nom
  } ; n = Sg ; p = P3 } ;
```

The substitutions of known arguments has been made (`Sg` of the type `Number`), the functions applied (`detNounPhrase, artdef, toCase`) and the table expanded. The only operations left are concatenation (`++`), projection (`.`) and table selection (`!`). Function application and table expansion are the corresponding evaluation procedures for *function definitions* and *pattern matching* - two abstraction mechanisms in the GF grammar formalism. Such mechanisms help the grammarian to work on a higher level of abstraction.

In the example above, the same expression

```
"the" ++ (CN_0.s ! Sg) ! Nom
```

is repeated three times. This can be optimized by using the flag `optimize=all_subs` in the grammar.

Compilation into the canonical form actually does more than partial evaluation. It also represents a grammar in a format adapted for easy usage in syntax editing implementation. Thus, grammar compilation fills in the gap between the abstract theory and the implementation.

2.2 Implementation

Gramlets generation scheme is shown in Fig. 22.

First the grammar files are loaded in the main GF. A special command produces the XML format of the canonical GF grammar form. After that we leave

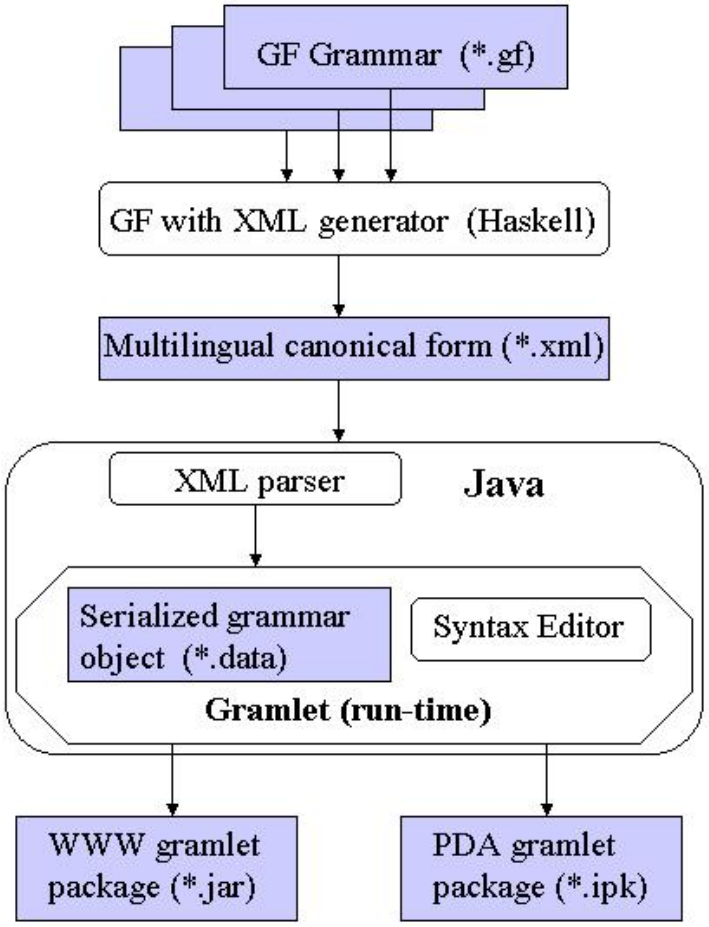

Figure 22: Gramlets production architecture. Files (shaded rectangles) are fed into and produced by the processing modules (rounded rectangles). Arrows indicate the information flow. The resulting run-time system is shown as an octagon containing the serialized grammar object and the syntax editor. The generated gramlet can be converted into packages for WWW and PDA.

the Haskell side and only use the XML output we got from it. Notice that the original GF grammar can be distributed into several files: one for abstract language-independent part, and one for each language represented. The XML canonical grammar representation is put into one output file, since it is not supposed to be read by the user but only by the automatic parser.

The next step is to convert the XML representation into Java grammar object representation. This is done by the XML parser written in Java. The reason we have the intermediate XML representation and do not produce the Java grammar object directly is the efficiency. The size of the automatically generated Java module to produce a grammar object was too big and, therefore, caused a run-time error. That is why we have chosen to generate the Java grammar object separately and then serialize it and store it in another file, so that the run-time GUI just have to read the Java grammar object from the file instead of creating it on the fly. With this approach it takes about 20 seconds to start a gramlet on Zaurus although some very big grammars (using the resource library) produce the error `out of memory` on Zaurus.

The GUI duplicates the functionality of the Java GUI Syntax Editor from section 1. However, it differs from the latter in two ways:

- It implements the syntax tree editing operations, while the Java GUI Syntax Editor just displays what has been sent to it by the main GF. On the other hand, the Java GUI Syntax Editor has richer functionality regarding both the layout options and the performed computations.

- The AWT Java GUI library is used for GUI controls on Zaurus instead of SWING because of limited JVM implementation.

Syntax tree editing operations like navigating the tree, adding and removing nodes are done using the Zipper structure [15] in the main GF written in Haskell. Java Gramlets we do not have a similar structure, since pointers (object references) mechanism is provided in this non-functional language. The elementary (without dependent types) type checking is done by ordinary statements `if-then-else` in the imperative editing procedures.

There are two GUI versions for Gramlets - one using AWT (for Zaurus) and one using SWING. The first one is not an applet due to some layout problems with the applet class on Zaurus. The second one is an applet.

The table below gives the figures on each of the Gramlet processing modules:

	XML generation(Haskell)	XML parsing	Canonical GF	GUI
Lines	163	750	2300	1750
Total	6100			

This code was written during half a year period (not full-time though) with three active project participants (not including the GF core written previously in

Haskell by Aarne Ranta). The XML generation in Haskell was done by Markus Forsberg. The Canonical GF classes in Java were written by Kristofer Johannisson. The author's part is the GUI and XML parser modules and also putting everything together to produce the final result.

The algorithms used are a direct translation from the corresponding Haskell modules (by Aarne Ranta) into Java. Each structure in the Canonical GF format is represented as a separate Java class, which gives around 40 classes most of which are rather trivial, while the same thing in Haskell takes just a couple of pages. Thus the Canonical GF in Java is a very big and slow structure, which leads to the efficiency problems (`out of memory` error on Zaurus with some bigger grammars).

This explosion effect was expected from the start and it is the reason why the direct re-implementation of the full GF in Java is infeasible. However, the Gramlets project is an interesting experiment on porting GF to an imperative language in principle. Java was chosen because of the portability issues, rich GUI library and the Unicode usage, which is important for multilingual grammars. This makes the work relatively straightforward and possible to do in a reasonable amount of time.

Another convenience brought by writing the whole program in Java is independent development. With Java GUI Syntax Editor a special XML protocol has been developed for communicating between Haskell and Java side. This protocol is only used for sending the results computed by the main GF system to the Java GUI for display. A special command shell to be sent in the opposite direction from Java GUI to the Haskell side was written. Thus, Haskell side and Java GUI side are highly dependent on each other and a modification of one of them in most cases requires corresponding updates in the other part. These complications are avoided in Gramlets, which only uses the XML output file from GF.

Further improvements are possible that can affect the efficiency. Most likely they have to do with reducing the Canonical representation structure and adjusting it to the imperative programming style.

The XML processing can be made more systematic by using parser generators and other techniques. This will make the code cleaner and more readable. However, since the XML processing is separated from the run time process this will not cause any improvement in the final result although it may speed up the intermediate step of a grammar object creation, which now takes around one minute for bigger grammars.

Even the GUI can be made richer or at least attaining the level of the Java GUI Editor. This of course will not help to solve the efficiency problem.

Bibliography

[1] D. Aspinall. The LEGO Proof Assistant. `http://www.dcs.ed.ac.uk/home/lego/`, 1999.

[2] B. Bringert. The transfer programming language. URL: `www.cs.chalmers.se/~aarne/GF/doc/transfer.html`, 2005.

[3] B. Bringert and A. Ranta. A pattern for almost compositional functions. In *Proc. International Conference on Functional Programming ICFP, Portland, Oregon, September*, pages 216–226, 2006.

[4] Björn Bringert. Embedded grammars. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, February 2005.

[5] C. Brun, M. Dymetman, and V. Lux. Document structure and multilingual authoring. In *INLG'2000, Mitzpe Ramon, Israël*, pages 24–31, 2000.

[6] O. Caprotti. WebALT! Deliver Mathematics Everywhere. In *SITE 2006, Orlando, USA*, 2006. URL: `webalt.math.helsinki.fi/content/e16/e301/e512/PosterDemoWebALT.pdf`.

[7] T. Coquand. An algorithm for type checking dependent types. *Science of Computer Programming*, 26:167–177, 1996.

[8] H.-J. Daniels. Multilingual syntax editing for software specifications. Master's thesis, Karlsruhe University, Karlsruhe, Germany, August 2005.

[9] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy. A structure-oriented program editor: a first step towards computer assisted programming. In *International Computing Symposium (ICS'75)*, 1975.

[10] M. Dymetman, V. Lux, and A. Ranta. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249, 2000.

[11] G. Gazdar, E. Klein, G. K. Pullum, and I. A. Sag. *Generalized Phrase Structure Grammars.* Blackwell Publishing, Oxford, 1985.

[12] Gramlets Development Team. Gramlets Homepage, 2003. URL: www.cs.chalmers.se/~markus/gramlets.

[13] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 233–248. Springer, 2002.

[14] T. Hallgren. Home Page of the Proof Editor Alfa. URL: www.cs.chalmers.se/~hallgren/Alfa, 2003.

[15] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[16] J.Khegai. Java GUI syntax editor for GF. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

[17] J. Khegai K. Johannisson, M. Forsberg and A. Ranta. From grammars to gramlets. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

[18] J. Khegai. Java GUI Syntax Editor manual. URL: www.cs.chalmers.se/~aarne/GF2.0/doc/javaGUImanual/javaGUImanual.htm, 2004.

[19] J. Khegai. GF IDE for GF 2.1. URL: www.cs.chalmers.se/~aarne/GF2.0/GF-Doc/GF_IDE_manual/index.htm, 2005.

[20] Janna Khegai, Bengt Nordström, and Aarne Ranta. Multilingual syntax editing in GF. In A. Gelbukh, editor, *CICLing-2003, Mexico City, Mexico*, LNCS, pages 453–464. Springer, 2003.

[21] P. Ljunglöf. Expressivity and Complexity of the Grammatical Framework, 2004. URL: www.cs.chalmers.se/~peb/pubs/p04- PhD-thesis.pdf.

[22] Z. Luo and R. Pollack. LEGO Proof Development System. Technical report, University of Edinburgh, 1992.

[23] L. Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 1994.

[24] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS 806, pages 213–237. Springer, 1994.

[25] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[26] G. Perrier. Interaction grammars. In *COLING, Saarbrücken, Germany*, pages 600–606, 2000.

[27] R. Power and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL 98*, Montreal, Canada, 1998.

[28] R. Power, D. Scott, and R. Evans. Generation as a solution to its own problem. In *INLG'98*, Niagara-on-the-Lake, Canada, 1998.

[29] Richard Power, Donia Scott, and Anthony Hartley. Multilingual generation of controlled languages. In *EAMT/CLAW-03*, Dublin, Irland, 2003.

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

[31] A. Ranta. The GF Resource grammar library, 2005. URL: `www.cs.chalmers.se/~aarne/GF/lib/resource/doc/01-gf-resource.html`.

[32] A. Ranta. GF Homepage, 2006. `www.cs.chalmers.se/~aarne/GF/`.

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

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

[35] M. Steedman. Combinators and grammars. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. D. Reidel, Dordrecht, 1988.

[36] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[37] K. van Deemter and R. Power. Multimedia document authoring using wysiwym editing. In *INLG-2000*, pages 15–19, Mitzpe Ramon, Israël, 2000.

[38] Jim Welsh, Brad Broom, and Derek Kiong. A design rationale for a language-based editor. *Software – Practice and Experience*, 21(9):923–948, 1991.

[39] Y. Bertot. The CtCoq System: Design and Architecure. *Formal Aspects of Computing*, 11:225–243, 1999.

[40] Sharp Zaurus. Sharp Zaurus Homepage, 2003. `www.myzaurus.com`.