Thesis for the Degree of Licentiate 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, December 2003

Language engineering in Grammatical Framework (GF)
Janna Khegai



Technical Report no. 30L
ISSN 1651-4963
School of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, Sweden, 2003

Language engineering in Grammatical Framework (GF)

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

**Abstract**

This thesis describes a number of practical experiments rather than theoretical investigations in the area of natural language processing.

The basis for the work presented is Grammatical Framework (GF). It is a very complex system, which comprises among other things a grammar formalism based on type theory and its implementation written in Haskell. GF is intended for high-quality machine translation (of INTERLINGUA type) in the restricted language domains.

The primary concern of this thesis is however limited to 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 a part of code for automatic generation of gramlets — pure Java programs with limited (compared to GF) functionality that can be run on PDA (Portable Device Assistants) and as applets in a browser.
- 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 restricted language domains) in Russian.

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

- Author level (end-user) — constructing sentences in natural languages.
- Grammarian level — building a grammar description, which is later used on the author level.

The last part of the thesis deals with a non-linguistic domain. In that experiment we try to apply functional parsing technique to the well-known problem of protein secondary structure prediction (bioinformatics).

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

**Acknowledgements**

I would like to thank my supervisor, Aarne Ranta for giving me a project in the first place and for careful guidance in carrying it out. From him I have learned a lot about the research field and beyond.

Thanks a lot to Arto Mustajoki and his colleagues from the Slavonic and Baltic Department at the University of Helsinki for fruitful discussions on Russian grammar. Also thanks to Mats Wirén from Telia Research, the discussion leader for his useful comments. Many thanks to my co-workers on this thesis - Kristofer Johannisson and Markus Forsberg and to the members of my advisory commitee - Bengt Nordström and Devdatt Dubhashi who donated their time and ideas to the project.

I am grateful to my former office mates: Angela Wallenburg, Wojciech Mostowski and Erik Kilborn for being both kind and helpful on many occasions. I also appreciate the efforts of many PhD students who have organized and participated in numerous social activities like parties, pubs, PhD cakes, skiing trips, brännboll matches, laserdome games etc., which I have enjoyed very much. Special thanks to Tuomo Takkula, Wojciech Mostowski, Boris Koldehofe and Alex Sinner for skiing lessons in Trysil.

My gratitude to all the 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, particularly my husband, Alexandre for constant technical, financial and moral support.

# Contents

# Chapter 1

# Introduction

The Grammatical Framework (GF) is a grammar formalism based on type theory. Together with implementation it forms a framework for performing various Natural Language Processing (NLP) tasks. In this chapter we will give a brief overview of the system. For a systematic description of GF we refer to [Ranar, Ran03].

## 1.1 The GF system

The 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 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 the language of framework with a grammar formalism — 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. Type system 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.

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 abstract and concrete part. Let us consider a simple Letter grammar. 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 for constructing a letter after the reserved word `fun`. The first function (rule) 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 semantical representation (meaning) of the domain.

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

```
param
  Sex = masc | fem ;
  Num = sg | pl ;

lincat
  -- linearizations for "Message" and "Ending" are omitted

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

lin
  -- linearization of MkLetter is omitted

  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} ;
```

Here we can see the correspondence between the abstract and the concrete part. Each category introduced in the abstract part has a linearization in the concrete part after the reserved word `lincat`), each function — after the reserved word `lin`. 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`.

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`.

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 fields' values `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 need to use the number and the sex fields' values of the argument `rec`. Then we can 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 ab-

stract part should be refined enough to take into account all possible grammar variations in different languages. For example, the English part does not really use the parameter `Sex`, since the word *dear* does not change. However, the two versions `NameShe` and `NameHe` of the same function are kept for the sake of the 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 command `include` is used whenever it is necessary to access the content of another file. Therefore, 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 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. Abstract syntax represents language independent properties of the described natural language domain.

Concrete syntax, represented by linearization rules introduces actual strings (words) from certain 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 the grammar books. Records remind of dictionary entries. For example, here is the description of the word *она* (*she*) in the Russian concrete syntax:

```
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 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 about 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 having the results of logical frameworks. Parsing is more difficult. The GF grammar formalism is stronger than context-free grammars. Parsing in GF consists of two steps:

- context-free parsing (a number of parsers are implemented including basic top-down, Earley, chart)
- postprocessing phase

The result produced by a context-free parser is further transformed by postprocessing, which mainly consists of argument rearrangements and consistency checking for duplicated arguments. Correspondingly, 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 that will be used by the postprocessor. *Profile* describes the mapping from the position of a rule argument in the tree (after postprocessing) 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 abstract syntax (semantic model). 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*, which means "very white snow". In some languages reduplication is used to form plural form [Lin95]. The expressive power of the GF grammar formalism permits to handle these phenomena known to be non-context-free [JM00].

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

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

we will get 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
Abstract part:

```
cat S; Aux
```

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

Concrete part:

```
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; }
```

## 1.2 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 we can distinguish between the different kinds of usages.

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

On the first level the user is only allowed to write documents within the pre-existing grammar set. The graphical user interface (GUI) provided by Java GUI Syntax Editor described in chapter 2.1 ensures that work on this level will not require any specific knowledge of GF.

On the next 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 according to 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. Chapter 3 is 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 transformation algorithms are hidden. Parsers for user grammars are generated automatically. Finally, an implementor is supposed to be a programmer with fair knowledge of linguistics in general and the GF system in particular. GF is a really inter-disciplinary project.

We are aiming at creating an Integrated Development Environment (IDE) for the GF language that contains tools for users on different levels. This document is a collection of more or less independent reports describing several projects

within the GF system reflecting the progress towards our goal. GF is constantly developing, so the system status described is subject to change.

## 1.3 Thesis overview

Chapter 2 consists of three parts all related to syntax editing in GF:

- Introductory section tries to address the theoretical issues behind the implementation.
- In the next section we look at GF as a multilingual authoring tool.
- Final section tells about implementing Gramlets — a specialized version of the GF syntax editor running on PDA and WWW.

Chapter 3 is devoted to resource grammars — a supporting library for the grammarian. Adding Russian to the resource library is the main result of this chapter.

Chapter 4 describes out attempt to use GF and Haskell for protein structure prediction.

The concluding chapter 5 contains a discussion of related work, results and future plans.

There are four appendixes:

- The command reference for Java GUI syntax editor described in section 2.1.
- Automatically generated test examples (in English and in Russian) from the resource grammar library described in chapter 3.
- The resource grammar library module `types` (Russian) containing the description of word classes and morphological parameters.
- The resource grammar library module `syntax` (Russian) with syntactic rules.

The last two appendixes are automatically translated from the GF grammar format into the latex format using the *gfdoc* tool.

# Chapter 2

# Syntax editing

The editing procedure in GF is strongly connected to the concept of interactive theorem proof construction in proof editors like ALF or Alfa[Mag94, Hal03].

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 2.1). The second is Gramlets (section 2.2), 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 formal 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.

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 build 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 reported. 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 underlying meta-theory.

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 [Coq96].

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 2.1.7). 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 2.1 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 improve the performance.

Syntax editing process is replacing (or refining or instantiating) the metavari-

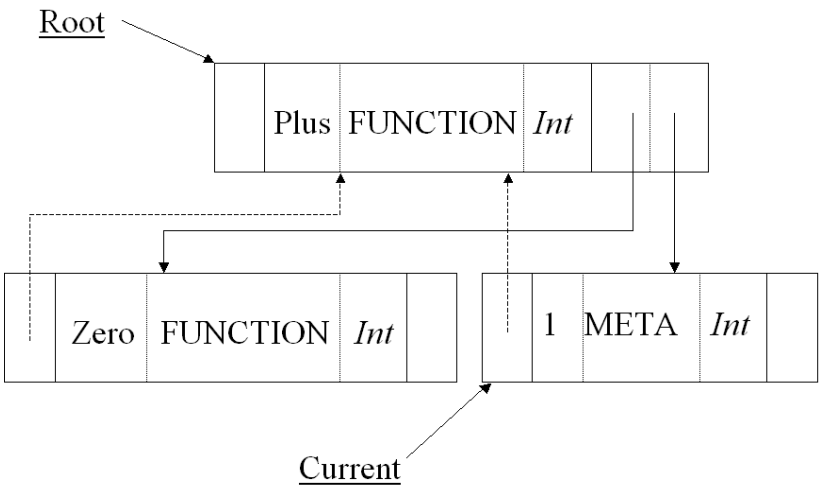


Figure 2.1: The abstract syntax tree for the expression $0+?_1$. The root contains the operation function `Plus`, which takes two arguments (children nodes) and returns the result if the type *Int*. One argument is the function `Zero` (with *Int* result), while the other is not yet known. Therefore, 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.

ables 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. 2.2 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. If there is more than one argument of this type several alternatives will be presented. 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. 2.2 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. 2.2 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 ->A* to remove the function while keeping 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. 2.2 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 Fig. 2.2 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*.

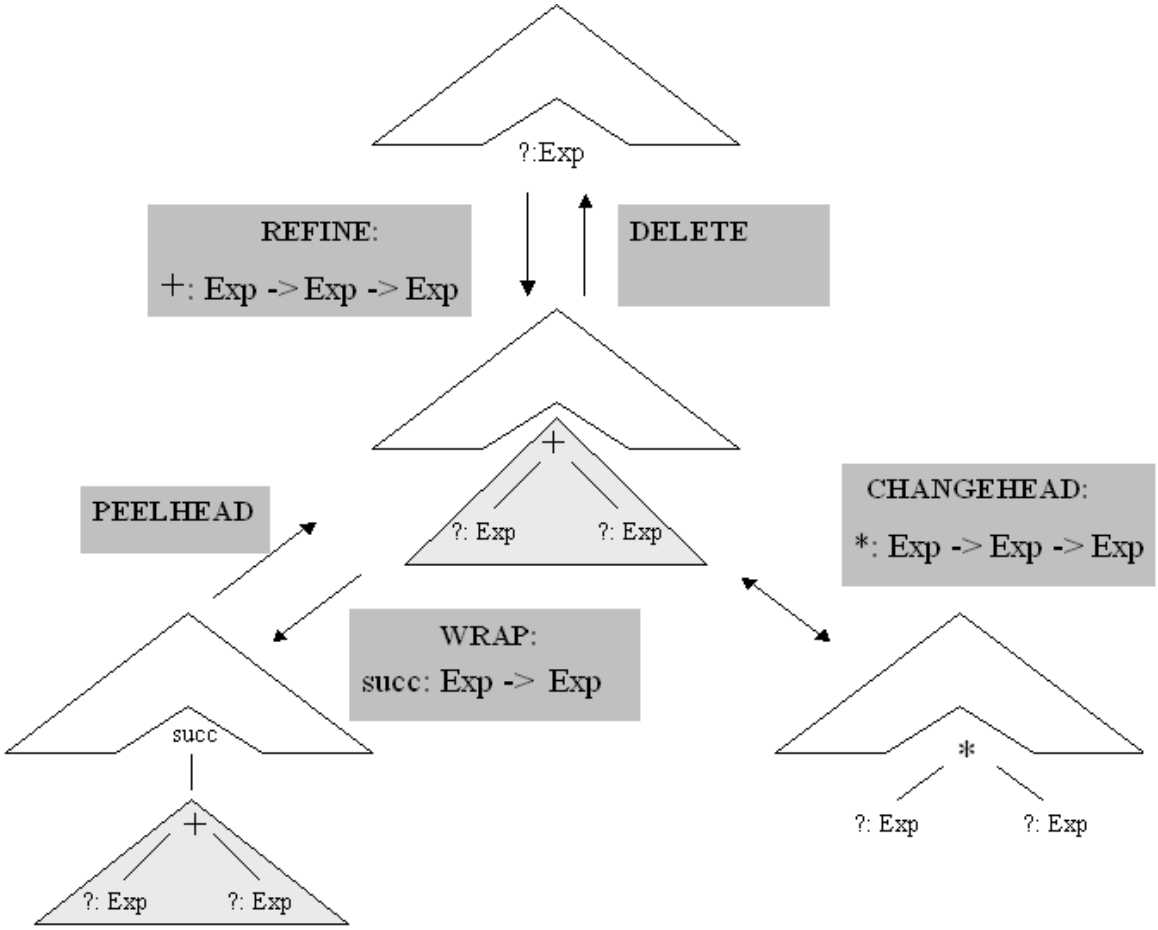


Figure 2.2: Examles of 5 basic types of syntax editing commands in GF. Syntax trees contain hidden parts (shown as roofs) and the focus nodes with its subtrees (if any). The original abstract syntax tree is the top tree. Arrows show possible refinement steps. The type of the editing step and the type signature of the refinement function are shown near the corresponding arrow.

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. The rest of the operations are more specific to GF. For syntax editing examples using the Java GUI editor see section 2.1.

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.

## 2.1 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 [KNR03, Khe02, J.K03].

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. 2.3). 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.

### 2.1.1 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. 2.4.

The main areas are:

- *Tree Panel* – displays the abstract syntax tree (AST) representation of the edited object.

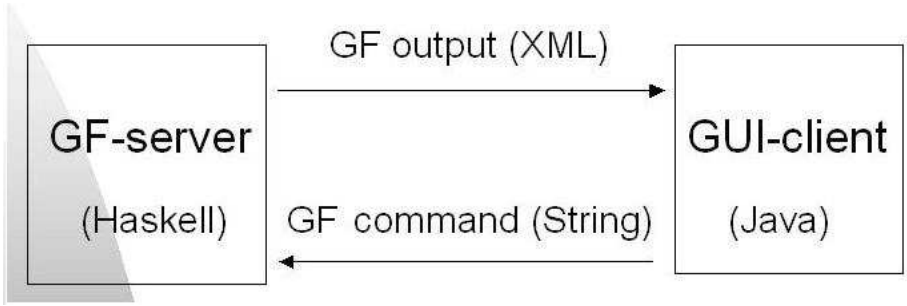


Figure 2.3: The communication between GF and Java GUI is performed according to the client-server architecture.

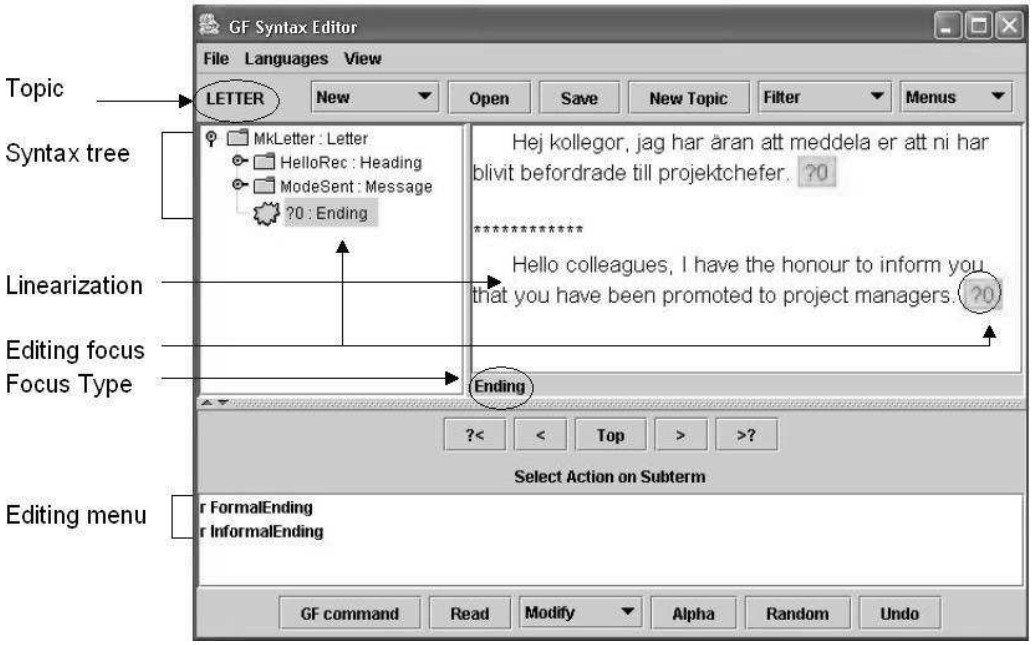


Figure 2.4: . Java GUI Syntax Editor's structure.

- *Linearizations Panel* – shows the linearizations corresponding to the AST in different languages. In Fig. 2.4 there are linearizations in Swedish and English.

- *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.

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. 2.4 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.

- *Focus Type* – specifies the syntax type of the editing focus. In Fig. 2.4 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 A.

### 2.1.2 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. 2.5). In Fig. 2.6 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.

### 2.1.3 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,


Figure 2.5: 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.

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. 2.7(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 2.1.4.

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 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. 2.7(b)).

Notice that the Swedish and French linearizations now contain the female

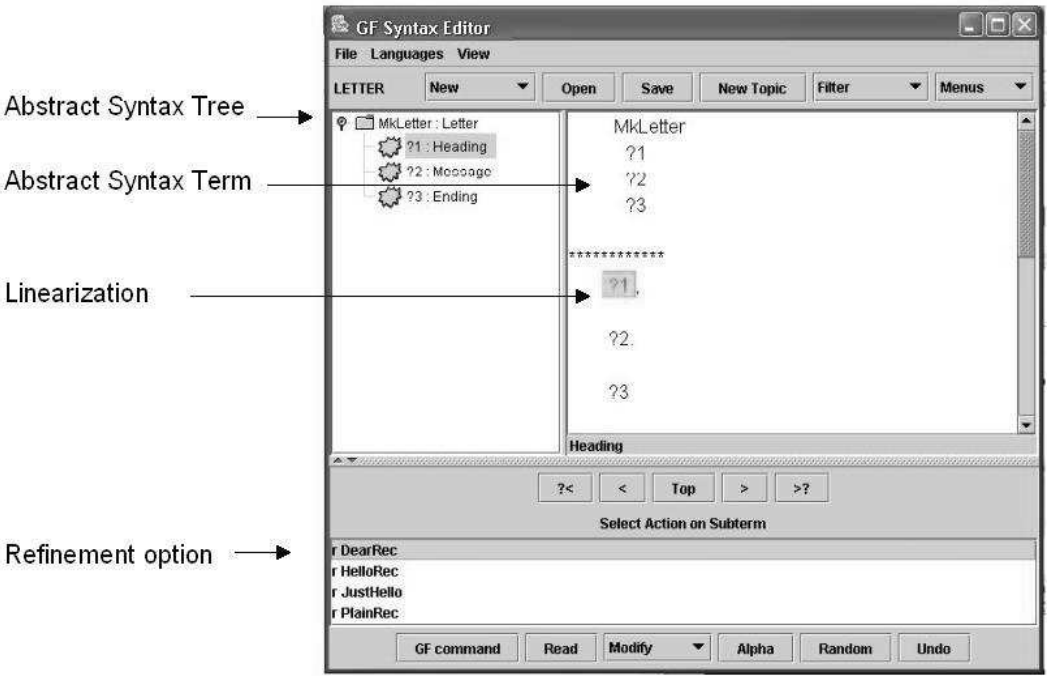


Figure 2.6: The Abstract Syntax tree represents the letter structure. The current editing focus, the metavariable ?1 is highlighted. The type of the current focus is shown below the linearizations area. The context-dependent refinement option list is shown in the bottom part.

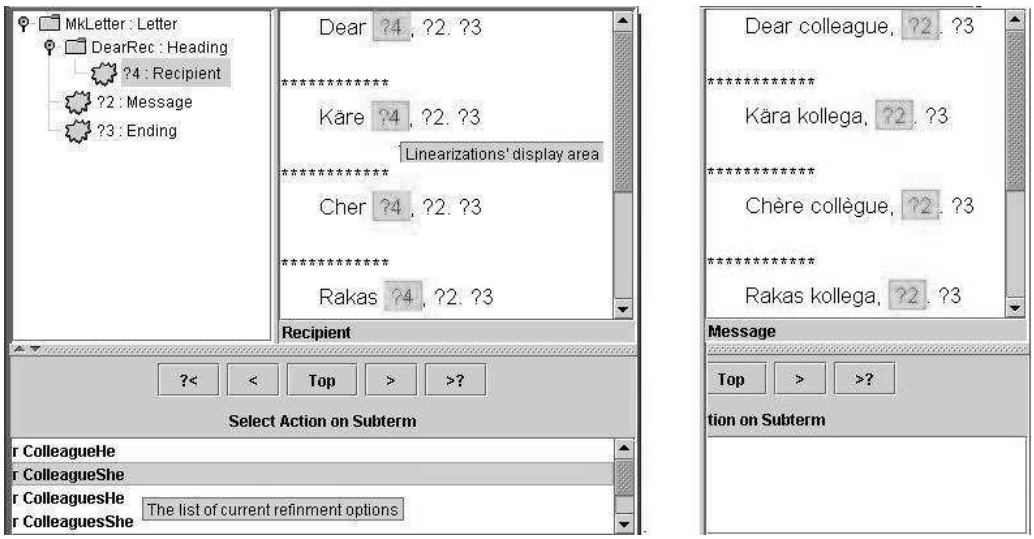


Figure 2.7: (a) The linearizations are now filled with the first word that corresponds to *Dear* expression in English, Swedish, French and Finnish. The refinement focus is moved to the Recipient metavariable. (b) The Heading part is now complete. The adjective form changes to the corresponding gender after choosing the recipient.

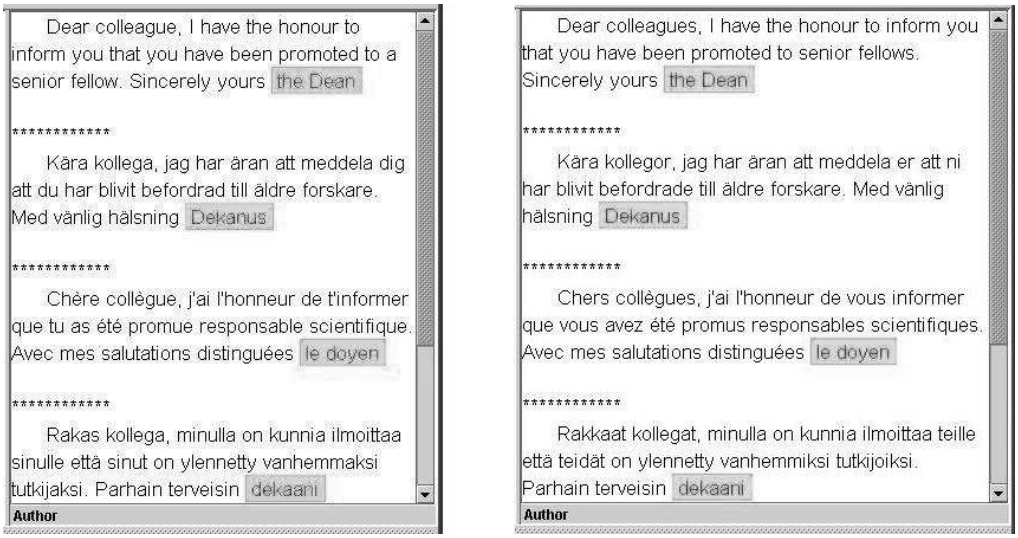


Figure 2.8: (a) The complete letter in four languages. (b) Choosing the plural male form of the Recipient causes various linguistic changes in the letter as compared to (a).

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. 2.8(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 to wrap them by functions.
- *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: $f : A \to A$ to remove them while keeping the argument subtrees. Can be seen as opposite to the wrap command.
- *d (delete)* – used on non-metavariables to delete them.
- *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

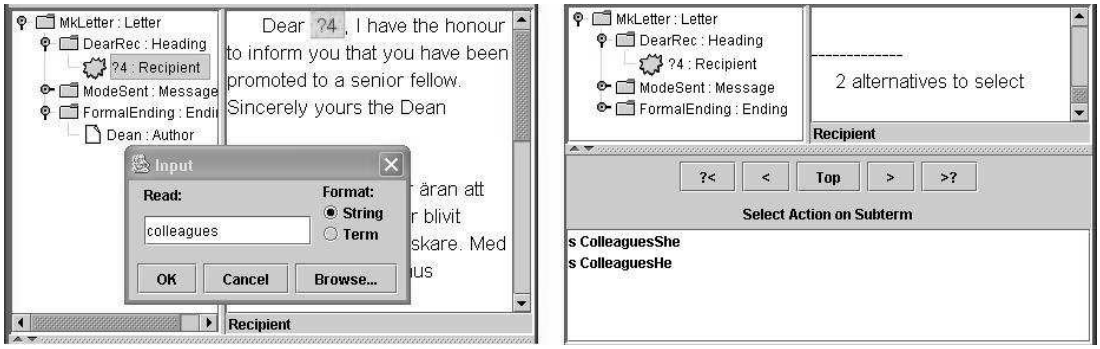


Figure 2.9: (a) A refinement step can be done by using the *Read* button, which asks the user for a string to parse (only in English in the present version).(b) When the parsed string in (a) is ambiguous GF presents two alternative ways to resolve the ambiguity.

to move the focus to the Header node in the tree and delete the old refinement. In Fig. 2.9(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. 2.9(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. 2.8(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.

### 2.1.4 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 while other things remain equal, including the document that is being edited. Fig. 2.10 shows the effect of adding Russian to the above example.

### 2.1.5 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

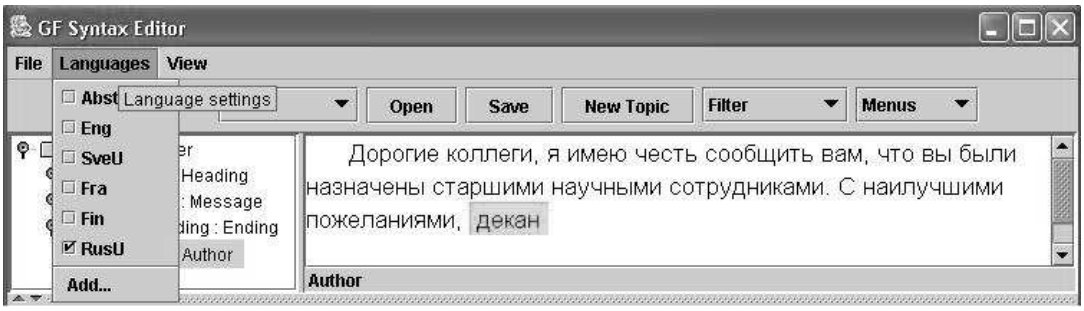


Figure 2.10: Now we are able to translate the letter into Russian.

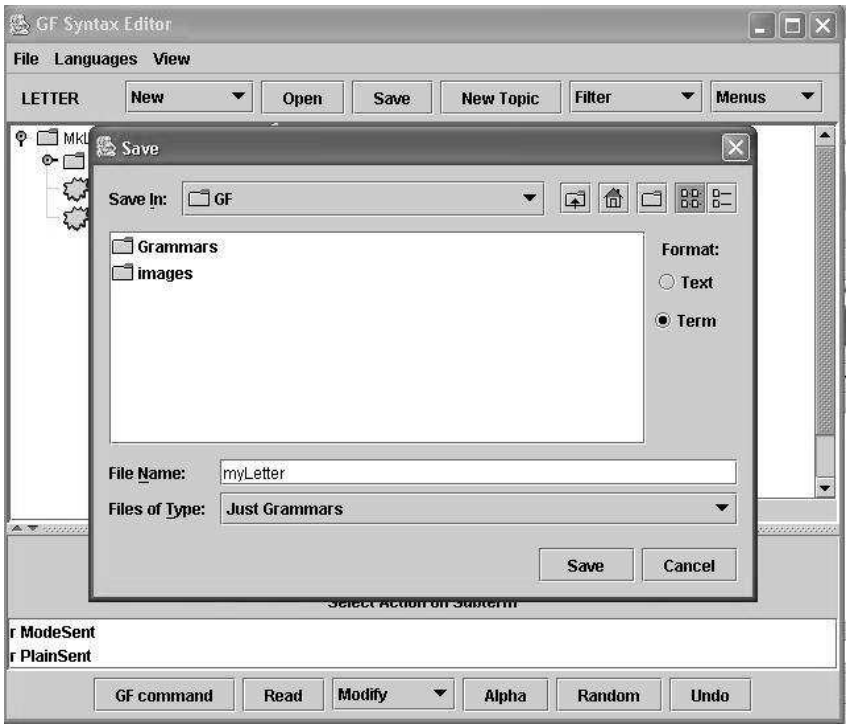


Figure 2.11: Pressing the Save button brings up a file chooser dialog.

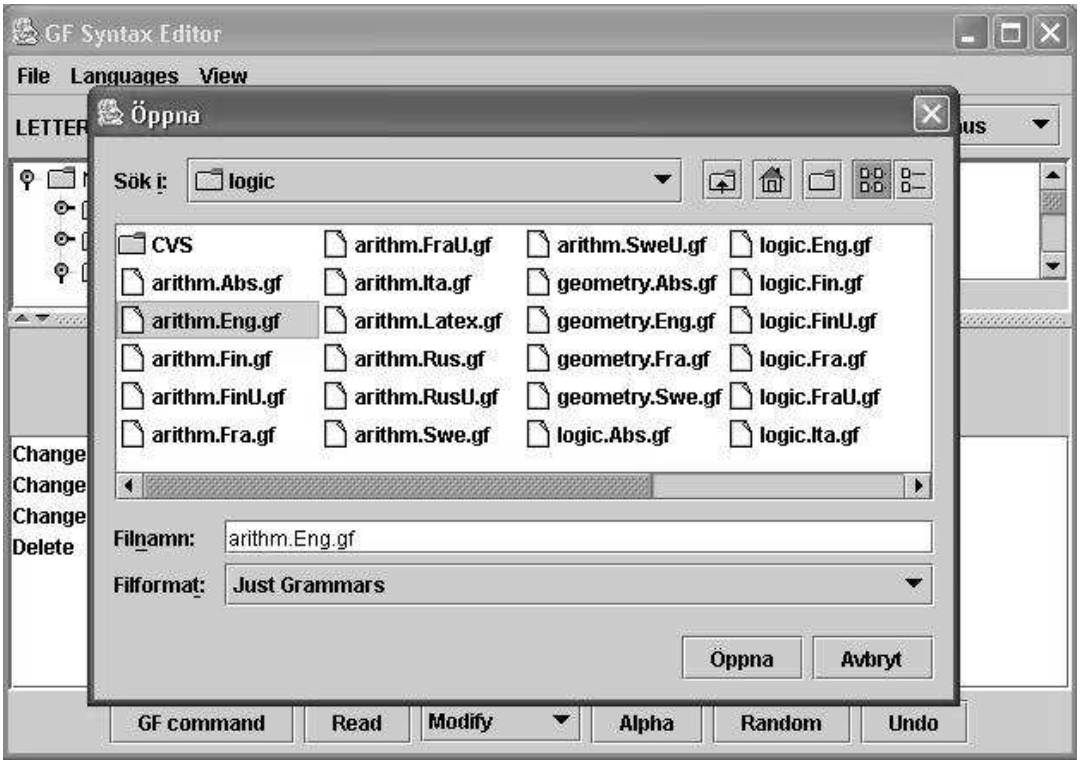


Figure 2.12: Pressing the *New Topic* button brings up an open dialog to choose a grammar file.

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. 2.11 we want to save the language-independent, abstract representation in the file named myLetter in the GF directory.

### 2.1.6 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

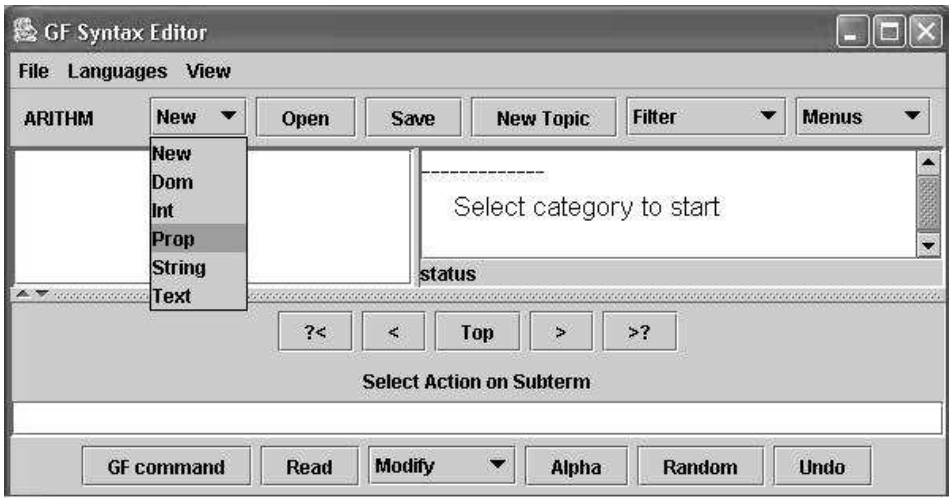


Figure 2.13: GF Syntax Editor after choosing the ARITHM (ARITHMETIC) grammar.

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. 2.12 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.

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. 2.13.

### 2.1.7 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. 2.14. 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 [Mag94, Hal03].

The current focus node *zero* can be wrapped, for example, with *succ* function.


Figure 2.14: 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.

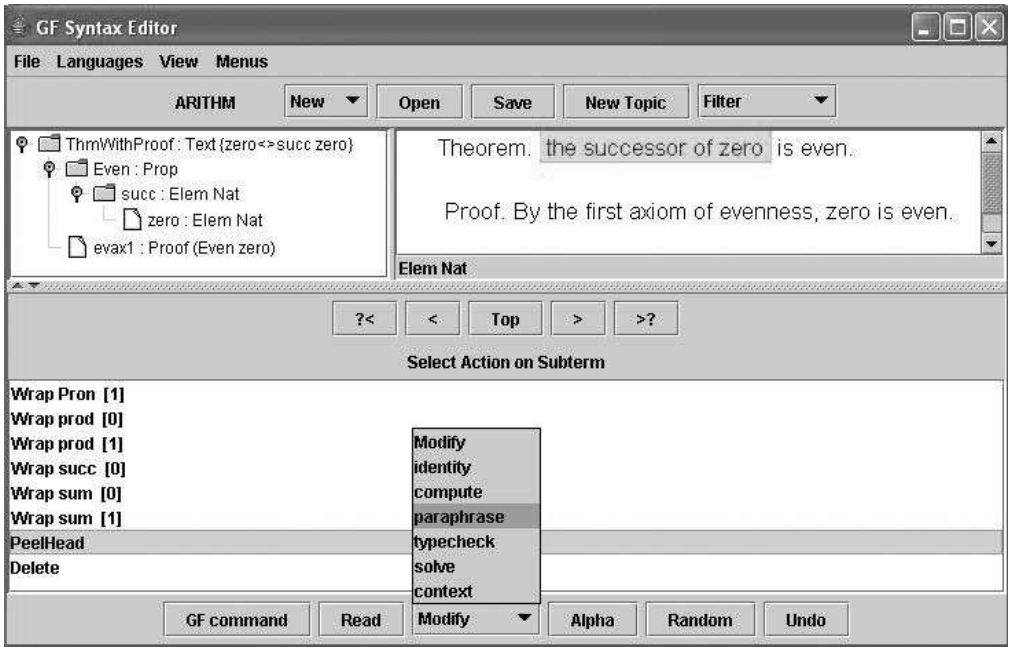


Figure 2.15: Constraint *zero <>?0* is produced by the type-checking procedure to assure that the theorem type is correct, namely, the proof part contains the proof of the stated proposition. *PeelHead* selection option undo the wrapping operation and restore the proposition in Fig. 2.14. This is only possible for functions that take only one argument and the types of the argument and of the result are the same. *Paraphrase* operation allows us to search for equivalent expressions.

The result is shown in Fig. 2.15. So, now instead of the statement *Zero is even* we 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 can not be proved within the arithmetic theory supported by the ARITHMETIC grammar. But what is even more interesting is that 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 are 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. 2.15 we can see such a constraint in the root node of the tree in brackets: *zero <>?0*. 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

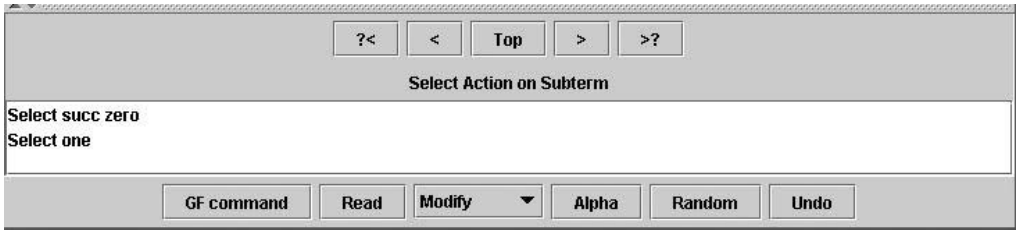


Figure 2.16: There are two versions (synonyms): *one* and *the successor of zero* to choose from.

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 the argument are of the same type. For example, if we chose to wrap with the *sum* function, which takes two arguments, we could not just peel the *sum* 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. 2.15. As the result of the paraphrase search we get two selection options, see Fig. 2.16. Namely, we can keep *the successor of zero* or use stylistically nicer *one* instead, see Fig. 2.17. 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. 2.17. It unfolds the function definition if any and computes the linearization of the result.

Dependent types allows us to even fill parts of a theorem by using the *solve* operation from the *Modify* menu, see Fig. 2.18. 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. 2.14.

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.

## 2.2 Gramlets: GF on-line and in the pocket

Gramlet is a Java applet/application with syntax editing functionality (described in section 2.1) for a specialized grammar [KJR03]. 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


Figure 2.17: Compute operation will unfold the definition of *one* giving *the successor of zero*.

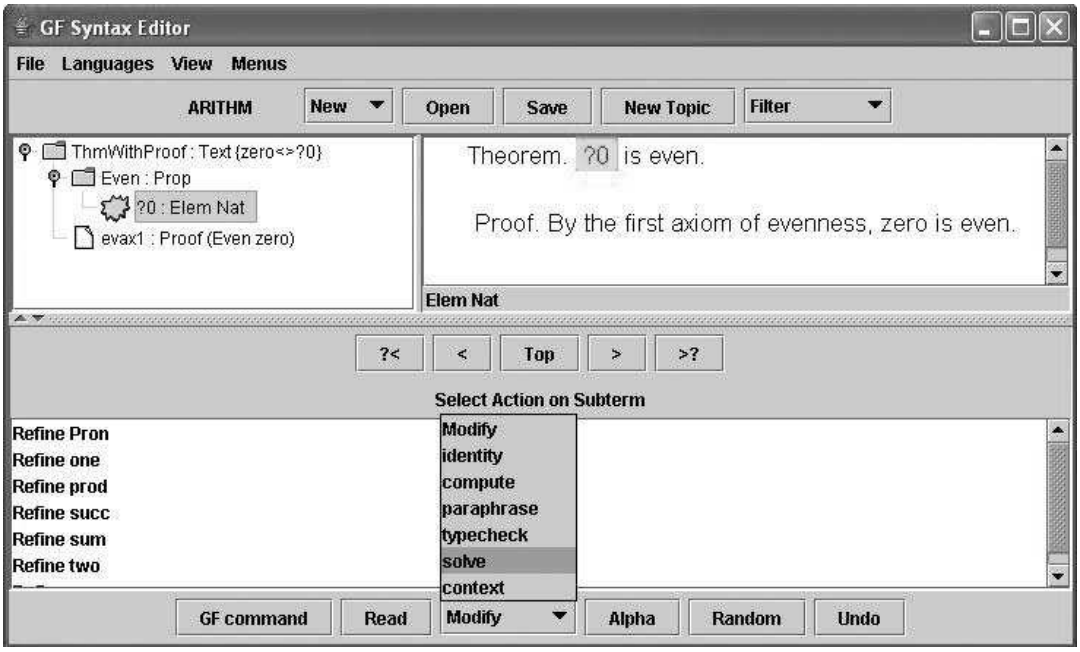


Figure 2.18: *Solve* operation will resolve the type constraint *zero <>?0* (in the root node) and fill the metavariable *?0*.

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 [Zau03], 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. 2.19 For on-line example visit the Gramlets homepage [Tea03].

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. 2.19 we can see the Health gramlet (based on the Health grammar given as an example in subsection 3.3.2), 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.

### 2.2.1 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. However, for the simpler computation the grammars in this notation must be normalized into so called canonical GF by type driven partial evaluation[Ranar]. 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

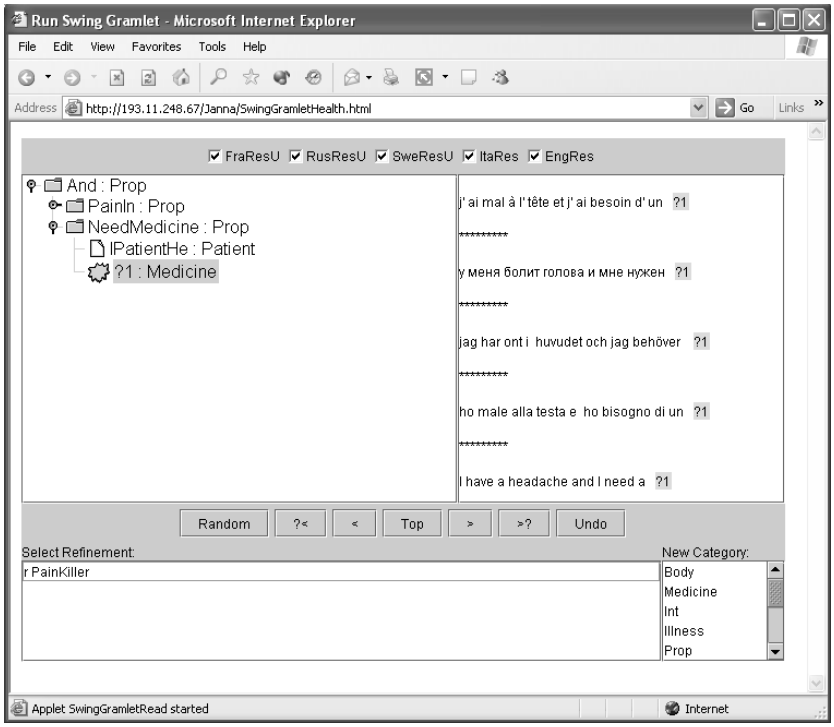


Figure 2.19: Health gramlet in the Internet Explorer browser.

regarded as functional programs (which they in fact are). In [KJR03] 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.

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.

### 2.2.2 Implementation

Gramlets generation scheme is shown in Fig. 2.20.

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 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 runtime error. That is why we have chosen to generate the Java grammar object

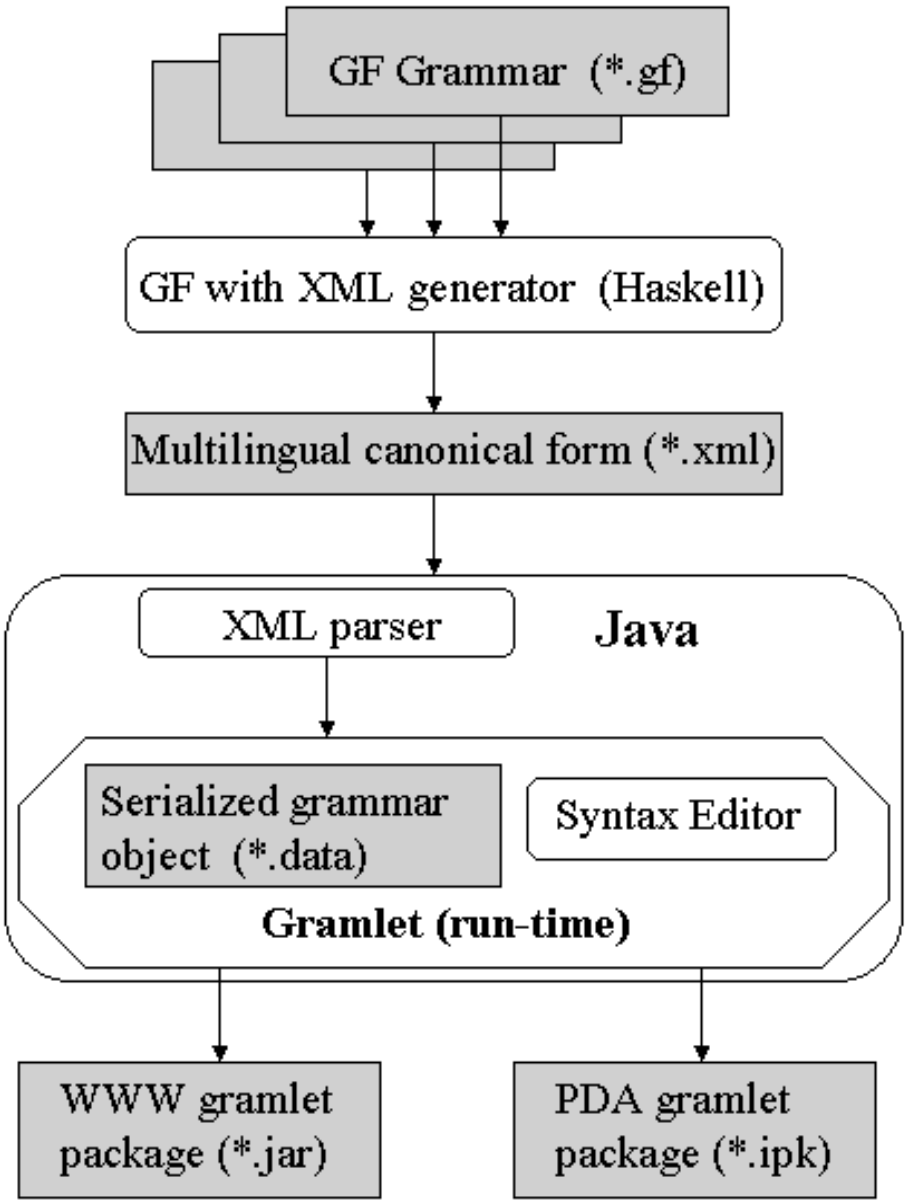


Figure 2.20: Gramlets production architecture. Files (shaded rectangles) are fed into and produced by the processing modules (rounded rectangles). Arrows indicate the information flow. The resulting run-time system is shown as an octagon containing the serialized grammar object and the syntax editor. The generated gramlet can be converted into packages for WWW and PDA.

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 2.1. 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 [Hue97] 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.

# Chapter 3

# Russian resource library

One of the strong 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 [Ran02] 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.

## 3.1 Resource grammar library structure

The file structure of a resource grammar layer corresponding to one language is shown in Fig. 3.1. Arrows indicate the dependencies among the modules.

The shadowed boxes correspond to the high-level modules. Ideally, the grammarian should be able to write an application grammar looking only at the shadowed modules. Resource and Predication modules consist of both language-independent Abstract part as well as Concrete implementation for every language. Consulting the Abstract part should be sufficient for writing an application grammar.

The paradigms module unlike the Resource and Predication modules only contains the Concrete part, since it is responsible for adding new lexical entries, which are specific for the language. However, it is considered to be high-level, since it provides an interface for adding the entries without the necessity to know

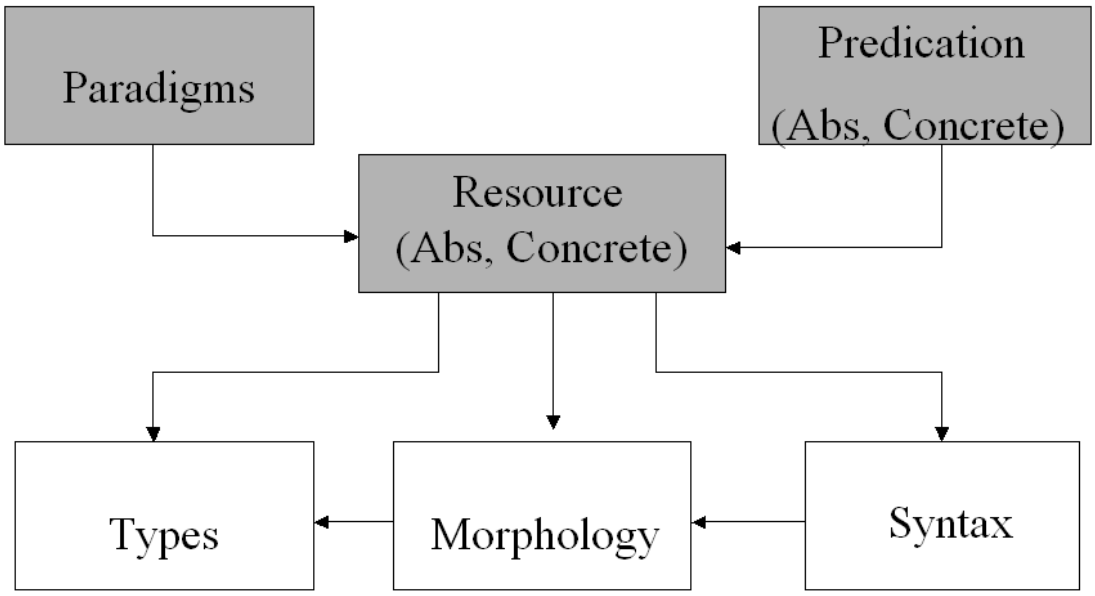


Figure 3.1: The resource grammar structure (main modules). One language layer. Shadowed boxes represent high-level of interface modules. White boxes represent low-level or implementation modules. Arrows show the dependencies.

the GF grammar language details. For examples, see subsection 3.2.2.

White boxes contain low-level functions specific for every language, whose type signatures are declared in the high-level modules. These files contain the actual implementation of the resource grammar using the GF grammar language.

The Types module defines morphological parameters and word classes of a language. However, it only includes those parameters that are not needed for analyzing individual words: such parameters are defined in Syntax modules. The type system of a language, in our experience, needs 50–100 lines of GF code. To get it right is a major design issue of a resource grammar, and the type system alone gives a good overview of the language.

The Morphology module contains the most common inflectional patterns for single words and lexical entries. The best approach of all seems to be to start with a morphology and build a lexicon incrementally as words get used in applications since typical application grammars are special-purpose grammars using special vocabulary. The content of the Morphology module can be also extracted from other sources. For example, the Swedish morphology module in the resource library is generated automatically from the code of another project called Open-Source Functional Morphology [Tea02]. Unlike the syntactic descriptions the morphological descriptions for many languages has been already developed in other projects. Considerable efforts can be saved by reusing of the existing code.

The heaviest module is the Syntax module, which covers linguistic rules on the higher sentence level. The resource syntax is an open-ended concept. We

have built it 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. However, it is not clear what belongs to a "complete syntax", in the way it is clear what belongs to a complete morphology.

What about a resource semantics? We follow a working principle in the resource grammar project: A *resource semantics*, defining the meanings of all words and syntactic structures, will not be built. Instead, semantics is given in application grammars.

Looking at the resource grammar structure we can see two possible starting points of designing a new grammar: top-down and bottom-up. The first approach goes from the high-level Resource grammar and fills the lower level modules starting from Syntax and eventually defining Types and Morphology modules. This approach is especially convenient if we are already familiar with the resource grammar structure or, even better, have written the resource grammar for another language. However, if we have no such experience it might be easier to move in the opposite direction: from Types to Morphology and then Syntax as the complexity grows filling the gaps in the Resource module along the way. In practice we use the mixture of these methods. The Resource module specifies what is needed in the low-level modules, but lower level modules need to be implemented first. It might be even more natural to design the resource grammar library having a certain application grammar in mind. Then the pieces needed for this grammar can be the first to work on and a natural testing domain appears.

The modules Predication and Paradigms are built on the basis of Resource and Morphology modules correspondingly, since they are not used by the rest of the modules. Many other modules can be derived from the modules shown in Fig. 3.1.

For testing the resource grammars we have written a small script that produces a text in a natural language using all the library functions. The output of the script allows us to discover grammatical errors by proof-reading (see appendix B).

## 3.2 Designing a resource grammar for Russian

Russian resource grammar was added after the similar grammars for English, Swedish, French and German. The language-independent modules representing the coverage of the resource library, therefore, were ready. The task was to localize them for Russian having the examples of similar modules for other languages. We have used several grammar reference books for Russian language [Pul84, She00, Wad00].

The Russian resource grammar layer has the structure shown in Fig. 3.1.

The table below gives several size parameters of the low-level or implementation modules:

| | Types | Morphology | Syntax | Total |
|---|---|---|---|---|
| Rules (+lexical items) | 40 | 52(+66) | 115 | 207(+66) |
| GF code lines | 75 | 1000 | 393 | 1468 |
| File length incl. comments | 265 | 1062 | 892 | 2219 |

The next table shows similar information for the high-level or interface modules:

| | Resource | Paradigms | Predication | Total |
|---|---|---|---|---|
| Rules | 120 | 47 | 22 | 189 |
| GF code lines | 129 | 152 | 27 | 299 |
| File length incl. comments | 245 | 362 | 39 | 646 |

This gives us about 400 rules on both levels. The size of one rule varies from 1 (often) to 35 (rare) GF code lines. Half of the file length is usually taken by the comments. To get a rough idea about the coverage of the resource grammar library see appendix B for automatically generated sentence examples in English used for proof-read testing of the resource grammar library.

Now we will present some rule examples from different Russian resource grammar library parts. The rules are written using the GF grammar language, whose syntax is close to that of functional programming languages. Let us start with the language-independent part or API. Here is the list of grammatical categories and grammar rules from the Abstract syntax module that will be useful for our final usage example.

First we need to define the grammatical categories - data types representing parts of speech and syntactic elements:

```
cat
  N ;      -- simple common noun,    e.g. "car"
  NP ;     -- noun phrase,           e.g. "John", "all cars"
  VP ;     -- verb phrase,           e.g. "switch the light on"
  S ;      -- sentence,              e.g. "John walks"
```

A grammar rule is defined as a function on the grammatical categories reflecting the functional programming style of the GF grammar language. Here is the well-known verb phrase predication rule taking a noun phrase (`NP`) and a verb phrase (`VP`) as arguments and returning a sentence (`S`) as a result:

```
fun
  PredVP : NP -> VP -> S ;                -- "John walks"
```

Now we will look at different language-specific modules of the Russian resource grammar and give some sample definitions from these modules.

### 3.2.1 Types module

The parameters defined in the Types module include well-known gender, number, person and case values. Parameter definitions start with the reserved word `param` and enumerate all the possible values separated by a vertical mark `|` :

```
param
  Gender  = Masc | Fem | Neut ;
  Number  = Sg | Pl ;
  GenNum  = ASg Gender | APl ;
  Case    = Nom | Gen | Dat | Acc | Inst | Prep ;
  Person  = P1 | P2 | P3 ;
  Animacy = Animate | Inanimate ;
```

There are complex values like `GenNum` that have more than one constituent: `Asg Gender`. Such parameters called *hierarchical* are combinations of other parameters. In case of `GenNum` we have a combination of `Number` and `Gender`, which is used for adjectives, where plural never makes gender distinction. `GenNum` reduces the two dimensional gender-number table by merging the plural forms into one value `APl`. In the first value `ASg Gender`, `Asg` denotes that the number is singular, followed by a `Gender` value, for example, `ASg Fem` or `ASg Masc`. Actually, in such complex parameters the first constituent called constructor is implemented as a function: $ASg : Gender -> GenNum$.

The common word classes description includes noun, adjective, verb and preposition.

Nouns (`CommNoun`) decline according to number and case. For the sake of shorter description these parameters are combined in the type `SubstForm`. Nouns moreover have inherent gender and animacy parameters that affect verb conjugation and adjective declension respectively:

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

The parameter `SubstFrom` is a combination of `Number` and `Case` with constructor prefix `SF`. `CommNoun` definition starts with a reserved word `oper`. The word `Type` between a colon and an equality sign indicates that we are defining a data type. The data type itself is basically a record of fields, separated by semicolons. Each field has a name and a type. For example, `anim` field has type `Animacy`. More complex fields can depend on parameters. For example, field `s` is a string (`Str`), which depends on `SubstForm` parameter. Thus, this field contains not just one string, but a table of strings. In order to use this string

we first need to use projection on `s` field and then we need to make a selection from the table by specifying the parameter value. For example if we have the common noun `mashina` (машина) corresponding to `car` in English in order to get the string машинам we use the expression `mashina.s !(SF Pl Dat)`, where dot denotes the projection operation and the exclamation mark denotes the selection operation, `SF Pl Dat` denotes that we need a plural form in dative case. See subsection 3.2.2 for the definition of `mashina`.

`Adjective` is a special category for the adjectives that are not used in the comparative or superlative form. Otherwise the degree parameter would be needed:

```
param
  AdjForm = AF Case Animacy GenNum ;
oper
  Adjective: Type = {s : AdjForm => Str} ;
```

The verb tense system in Russian consists of the present, the past and the future. The verb in the past tense conjugates according to the parameter `GenNum` while in the present and future - according to the parameters `Number` and `Person`. The verb mood can be infinitive, imperative, and indicative. Indicative form has different tense forms while imperative, infinitive and subjunctive modes do not. Imperative takes into account the number of the subject, subjunctive conjugates according to the parameter `GenNum` while infinitive has no variation at all.

```
param
  VTense   = VPresent Number Person | VPast GenNum |
             VFuture Number Person ;
  VerbConj = VIND VTense | VIMP Number Person | VINF | VSUB GenNum ;
  VerbForm = VFORM Voice VerbConj ;
oper
  Verbum : Type = { s: VerbForm => Str ; asp : Aspect };
```

For writing an application grammar one usually does not need the whole inflection table, since each verb is used in a particular context that determines some of the parameters (`Tense` and `Voice` while `Aspect` is fixed from the beginning) for certain usage. So we define the type `Verb`, that has these parameters fixed. The conjugation parameters left (`Gender, Number, Person`) are combined in the type `VF`:

```
cat
  Verb ;
param VF =
  VFin  GenNum Person | VImper Number Person | VInf | VSubj GenNum;
oper
  Verb : Type = {s : VF => Str ; t: Tense ; a : Aspect ; v: Voice} ;
```

Prepositions are just strings:

```
oper
  Preposition = Str ;
```

### 3.2.2 Morphology and Paradigms modules

Having the inflection patterns as predefined functions we can easily add new lexical entries. Common inflections patterns are described as functions that take one or more string arguments and form the corresponding lexical entry.

For example, the function below describes the inflectional table for feminine inanimate nouns ending with *-a* in Russian. For nouns such operations take the root of the word as a string (`Str`) argument and returns the noun represented by the type `CommNoun`. Six cases times two numbers gives us twelve forms plus two inherent parameters: `Animacy` and `Gender`:

```
oper
  aEndInanimateDecl: Str -> CommNoun = \golov -> { s  = table

 { SF Sg Nom => golov+"а";
   SF Sg Gen => golov+"ы";
   SF Sg Dat => golov+"е";
   SF Sg Acc => golov+"у";
   SF Sg Inst => golov+"ой";
   SF Sg Prepos => golov+"е";
   SF Pl Nom => golov+"ы";
   SF Pl Gen => golov ;
   SF Pl Dat => golov+"ам";
   SF Pl Acc => golov+ "ы ";
   SF Pl Inst => golov+"ами";
   SF Pl Prepos => golov+"ах"
   } ;
    g = Fem ;
    anim = Inanimate } ;
```

where `\golov` is a $\lambda$-abstraction, which means that the function argument of the type `Str` will be denoted as `golov` in the definition. The table structure is straightforward: one line correspond to one table parameter value with `=>` separating the parameter value and the corresponding result. Plus sign denotes string concatenation.

For a higher level access to this function we define another function with a mnemonic (in Russian though using Latin alphabet) name `nGolova` in the Paradigms module:

```
oper
   nGolova:  Str -> N; -- feminine,inanimate,ending with голов-а
   nGolova = aEndInanimateDecl ;
```

The type signature together with the comment is the only thing the user of the resource grammar needs to know to make a decision of using the function `nGolova`. The implementation details are hidden. The type signature from the Paradigms module, therefore, provides a higher-level access to the resource morphology. For example, we can define the word `mashina` (*машина*) corresponding to the English word *car*. Машина is a feminine, inanimate noun ending with *-а*. Therefore, a new lexical entry for the word *машина* can be defined by:

```
   oper
    mashina = nGolova "машин";
```

The advantage of such higher level access becomes especially significant with more complex parts of speech like verbs. Here is the definitions of the verbs corresponding to `equal` and `divide` in English using the paradigms function `verbDecl`, which takes five strings coresponding to the basic forms and two extra parameters: `Aspect` and conjugation type. Aspect can be either `Imperfective` or `Perfective` in Russian. There are two main verb conjugation types in Russian: the `First` and the `Second`:

```
   oper
    verbRavnjat = verbDecl Imperfective First
                 "равня""ю""равнял""равняй""равнят";
```

If we unfold the `verbDecl` function we get a rather complex definition:

```
oper
  verbDecl: Aspect -> Conjugation -> Str -> Str ->
            Str -> Str ->Str -> Verbum =
   \a, c, del, sgP1End, sgMascPast, imperSgP2, inf -> case a of
{  Perfective  =>  case c of  {
     First  => mkVerb (perfectiveActivePattern inf imperSgP2
                (presentConj1 del sgP1End) (pastConj sgMascPast))
                (pastConj sgMascPast);
     Second => mkVerb (perfectiveActivePattern inf imperSgP2
                (presentConj2 del sgP1End) (pastConj sgMascPast))
                (pastConj sgMascPast)
} ;
   Imperfective  => case c of {
     First => mkVerb (imperfectiveActivePattern inf imperSgP2
```

```
              (presentConj1 del sgP1End) (pastConj sgMascPast))
              (pastConj sgMascPast);

     Second => mkVerb (imperfectiveActivePattern inf imperSgP2
              (presentConj2 del sgP1End) (pastConj sgMascPast))
              (pastConj sgMascPast)
   }
};
```

which takes aspect and the conjugation type together with five basic forms as arguments. According to the parameters we get a table with four cases altogether described by the `case` declarations. In each case the result is using another function named `mkVerb` with the corresponding arguments.

Such unfolding process can be continued further on. However, knowing the implementation details is not necessary for defining a verb entry , since the `verbDecl`' arguments can be easily decided using the comments in the module Paradigms having the actual verb to be defined. That is why the function `verbDecl` is considered to be a high-level function although its implementation looks quite complex and uses a lot of other low-level functions describing various morphological rules. Using such low-level functions directly would be difficult without getting into the technical implementation details. High level Paradigms access saves time and effort adding new lexical items.

### 3.2.3 Syntax module

Syntax module contains syntactic rules for forming syntactic parts `NounPhrase (NP)`, `VerbPhrase (VP)` and `Sentence (S)`. Unlike morphology entries such elements can contain more than one word and have some internal structure.

A noun phrase usually serves as a subject or an object in a sentence. For example, in *Mary and John love long walks.* both *Mary and John* and *long walks* are noun phrases. The first one serves as a subject, while the second one - as an object. A noun phrase comprises a noun or a personal pronoun. `NP` unlike a `CommNoun` entry from the lexicon has a fixed number and person parameter. It also has an extra parameter `pron` of the type `Bool` stating if the noun phrase is expressed by a pronoun, since Russian personal pronouns can act as noun phrases and have different forms than nouns:

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

where

```
param
  PronForm = PF Case AfterPrep Possessive;
```

is taken from the module Types. Here `Case` is a usual noun parameter, while `AfterPrep` and `Possessive` are used only for personal pronouns indicating, which form should be used in case there is a preposition before the noun phrase or the noun phrase has a possessive meaning. (Possessive pronouns are regarded as another morphological form of personal pronouns, since they are inflected just as personal pronouns according to the parameters `Case, Number` and `Gender`).

Verb phrases are discontinuous, namely, the parts of a verb phrase are:

- `s` — an inflected verb,
- `s2` — verb adverbials (such as negation),
- `s3` — a complement.

This discontinuity is needed in sentence formation to account for word order variations.

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

The sign `**` denotes that a `VerbPhrase` comprises all the `Verb`'s fields plus additional fields in the brackets. For example, let us consider the sentence *Я не вижу машину* (*I do not see the car*). Here *не вижу машину* (*do not see the car*) is a verb phrase, where *вижу*(*see*) together with some extra fields not shown in the resulting string has the type `Verb`. *не*(*do not*) is the value of the field `s2`. The field `negBefore` is True, since the negation goes before the verb (see the rule below). The field `s3` contains a table representing *машину*(*the car*). However, all the values in the table are the same in this example.

A non-trivial example of such a table can be *Машина - старая* (*the car is old*). Here the complement field `s3` of the verb phrase is formed by an adjective *старый* (*old*), which form depends on the gender and the number of the subject *машина* (*the car*). This agreement is taken care of in the traditional $S \to NPVP$ rule `PredVerbPhrase (PredVP)`:

```
oper
  predVerbPhrase : NounPhrase -> VerbPhrase -> Sentence =
    \Ya, tebyaNeVizhu -> { s =
       let
       { ya = Ya.s ! (mkPronForm Nom No NonPoss);
         ne = tebyaNeVizhu.s2;
         vizhu = tebyaNeVizhu.s ! VFin (gNum Ya.g Ya.n) Ya.p;
         tebya = tebyaNeVizhu.s3 ! Ya.g ! Ya.n
       }
       in
```

```
      if_then_else Str tebyaNeVizhu.negBefore
       (ya ++ ne ++ vizhu ++ tebya)
       (ya ++ vizhu ++ ne ++ tebya)

  } ;
```

where the sign ++ denotes concatenation. The function `if_then_else` shows how the value of the parameter `negBefore` from `VerbPhrase` affects the word order in a sentence.

### 3.2.4 High-level modules

The high-level modules are written on top of the low-level modules. Therefore, they do not extend the coverage of the resource grammar, but just provide a more convenient interface for the user. We have already mentioned the Paradigms module in subsection 3.2.2. Now let us take a look at the concrete part of the Resource module and Predication modules.

We discussed the abstract part of the resource module or API in the beginning of the section 3.2. The concrete resource module put together the language-independent API with an implementation for a specific language. Thus the functions and categories declared in the Abstract part get the concrete definitions (linearizations) in the Resource module using the operations predefined in the Syntax module:

```
lincat
  N      = CommNoun ;
      -- = {s : SubstForm => Str ; g : Gender ; anim : Animacy } ;
  NP     = NounPhrase ;
      -- = { s : PronForm => Str ; n : Number ; p : Person ;
      --     g: Gender ; anim : Animacy ;  pron: Bool} ;
  VP     = VerbPhrase ;
      -- = Verb ** {s2 : Str ; s3 : Gender => Number => Str ;
      --            negBefore: Bool} ;
  S      = Sentence ;
      -- = {s : Str} ;
lin
  PredVP = predVerbPhrase ;
```

The language-specific type definitions from low-level modules are put as comments starting by the sign -- for reference purposes. Actually, the user is supposed only to look at the abstract resource module.

The Syntax module contains only the most basic and, so to say, minimal operations, which are widely applicable. More complex macros derived from the rules from the syntax modules are however placed in a separate module called

predication library. For example, constructions with transitive verb like *John loves Mary* are very common. They consist of a Subject (*John*), a Verb (*loves*) and an Object (*Mary*). However, the classical rule `PredVP` expects only two arguments: a noun phrase and a verb phrase. Therefore, we can define the function `predV2` that takes three arguments:

```
oper
  predV2 : TV -> NP -> NP -> S ;    -- "John loves Mary"
  predV2 = \F, x, y -> PredVP x (PosTV F y) ;
```

whose implementation uses the classical rule `PredVP`. The resource grammar library user nevertheless does not need to know the implementation. All he needs to use the rule is a type signature with a usage comment (first line). This high-level type signatures are put in the abstract or language-independent part of the Predication library. The concrete implementation (second line) is put in the concrete part and again is not supposed to be consulted by the user. In this way the predication library provides a high-level access to the resource grammar just as abstract Resource (API) and Paradigms modules.

## 3.3 Application grammar examples

### 3.3.1 Arithmetic grammar: the resource and the non-resource version

The purpose of the first example is comparing two equivalent grammars written with and without the resource grammars library. We consider some fragments from a simple arithmetic grammar, which allows us to construct statements like *one is even* or *the product of zero and one equals zero.*

The abstract part of the resource and non-resource versions are the same and describe the meaning captured in this arithmetic grammar. This is done by defining first some categories:

```
cat
  Prop ;       -- proposition
  Dom ;        -- domain of quantification
  Elem Dom ;   -- individual element of a domain
```

The category `Prop` corresponds to the grammatical sentence. The other two categories define the domain of the arithmetic grammar and its elements. There are also some functions:

```
fun
  zero : Elem Nat ;                    -- zero constructor
  one  : Elem Nat ;                    -- one constructor
```

```
  Even  : Elem Nat -> Prop ;            -- evenness
  Prime : Elem Nat -> Prop ;            -- primeness

  EqNat : (m,n : Elem Nat) -> Prop ;    -- equality statement
  prod : (m,n : Elem Nat) -> Elem Nat ; -- product
```

The difference between the resource and non-resource versions appears in the concrete part of the arithmetic grammar. In the non-resource version we need to define the linearization types of the categories from scratch:

```
lincat
  Dom = {g : Gen; s : Case => Str } ;
  Prop = {s : Str} ;
  Elem = {g : Gen; s : Case => Str} ;
```

All we have to do in the resource version is to use the predefined categories:

```
lincat
  Dom = N ;
  Prop = S ;
  Elem = NP ;
```

Using the resource library we can define the functions Even and Prime in the following way:

```
oper
   Even = predA1(AdjP1(adj1Star "чётн"));
   Prime = predA1(AdjP1 (adj1Molodoj "прост"));
```

Here `predA1, AdjP1, adj1Molodoj` and `adj1Star` are taken from the resource library. The functions `adj1Molodoj` and `adj1Star` describe the lexical inflection patterns while the rest work on the syntactic level.

In the non-resource version we have to take care of the lexical inflection tables ourselves:

```
lin
   Prime n = \n -> { s = case n.g of
                  neu => n.s !  nom ++ ["- простое"];
                  fem => n.s !  nom ++ ["- простая"];
                  masc => n.s !  nom ++ ["- простой"] }
   Even n = \n -> { s = case n.g of
                  neu => n.s !  nom ++ ["- чётное"];
                  fem => n.s !  nom ++ ["- чётная"];
                  masc => n.s !  nom ++ ["- чётный"] }
```

Notice that the functions `Even` and `Prime` are very similar: the inflection patterns of the corresponding adjectives differs only in masculine gender. Therefore, we have to define basically the same thing twice to reflect this slight grammatical difference. This sort of redundancy increases as the size of the grammar grows.

This problem is less disturbing in the resource case, since the inflection differences are already taken into account by choosing the different functions `adj1Star` and `adj1Molodoj` for forming the right forms for the respective adjectives. Therefore, scaling up the grammar does not lead to the explosion of its size.

The functions `zero` and `one` declared in the abstract part are linearized as follows using the resource library functions:

```
oper
  zero     = DefOneNP (UseN nol) ;
  one      = DefOneNP (UseN edinica) ;
```

The functions `nol` and `edinica` represent lexical entities defined in the morphology module. The function `DefOneNP` is used to convert a common noun into a noun phrase, since both `zero` and `one` return the noun phrase result although these noun phrases only consist of one word. Similarly the function `UseN` is used to convert a noun lexical entry into a common noun in order to keep the types compatible. Such compatibility is important if we want to use the resource grammar library. This requires from the grammarian to be familiar with the type system of the grammar library as well as the various function signatures and can considerably slow down the work in the beginning.

In the non-resource case we have to define them as part of the arithmetic grammar. For every noun *zero, one* and *product* we need to define an inflection table like:

```
oper
   one = {g = masc ; s = table {
    nom => "единица";
    gen => "единицу";
    dat => "единице";
    ins => "единицей"} };
```

doing the job of the morphology module. Notice that for the purpose of the arithmetic grammar we describe the inflectional table only partially (not all of the possible forms are present). The partiality of such ad hoc definitions makes them unattractive for possible future reuse, since the missing forms can be needed for another grammar.

```
oper
  prod = appFunColl (funGen proizvedenie) ;
```

`proizvedenie` is a lexical entry from the morphology module. In general the functions starting from a lower case letters are taken from the language specific syntactic module, while the functions staring with a capital letter are taken from the language-independent API. `funGen` forms the function *the product of*, which `appFamColl` applies to two arguments joined by the conjunction *and*: for example, *the product of one and zero*.

The function `prod` is probably easier to understand (and to write) than its resource counterpart having that `product` is a morphological function similar to `one` above:

```
oper
   prod m n = { g = neu; s =table {
                cas => product!cas ++ m.s!gen ++ "и"++ n.s!gen }} ;
```

However, it contains low-level operations like `., !, ++`, while the resource version only uses function application.

The most complex function in the resource version is `EqNat`. See subsection 3.2.2 for `verbRavnjat`'s definition). The predicate `predV2` was defined in subsection 3.2.4. `mkTV` is from Paradigms module takes a `Verb`, a `Preposition` and a `Case` and returns a transitive verb (`TV`).

```
oper
  EqNat    = predV2 ravnjatsja ;
  ravnjatsja : TV = mkTV (extVerb verbRavnjat passive present)
                    nullPrep dative ;
```

The non-resource version has a similar structure separating the verb definition from the rest of the function:

```
oper
   EqNat m n = {s = m.s !  nom ++ equal !  m.g ++ n.s !  dat} ;
   equal:  Gen => Str = table {masc => "равен"; fem => "равна"} ;
```

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 may help to speed up, since the resource functions for adding new lexical entries are relatively easy to use.

Syntactic rules from the library 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.

### 3.3.2 Health grammar in five languages

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

We will consider a fragment of the Health grammar - a small grammar written using the resource grammar library in English, French, Italian, Swedish and Russian. It allows us to say phrases like *she has a cold and she needs a painkiller*. For this purpose we need the following categories and functions:

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

The category `Prop` denotes complete propositions like *she has a cold*. We also have separate categories for smaller parts like `Patient`, `Medicine` and `Condition`. An object of some type can be formed by applying the function that returns an object of that type. For example, to get an object of the type `Prop` one can use the function `BeInCondition`. This function takes two arguments of the types `Patient` and `Condition` correspondingly and returns the result of the type `Prop`. The abstract syntax determines the class of statements we are able to build within the grammar. For example, using the functions above we can form the phrase *she has a cold* by combining three functions:

```
BeInCondition ShePatient CatchCold,
```

where `ShePatient` and `CatchCold` take no arguments and return objects of the types `Patient` and `Condition`, which in turn are used as arguments to the function `BeInCondition`. The final result is the desired proposition.

The concrete part shared among the languages are all the categories and three functions that can be written entirely by API functions:

```
lincat
 Patient       = NP ;
 Condition     = VP ;
```

```
 Medicine      = CN ;
 Prop          = S ;
lin
 And           = conjS ;
 ShePatient    = SheNP ;
 BeInCondition = PredVP ;
```

Exactly the same rules work for all five languages, which makes the porting trivial. However, this is not always the case.

The word *painkiller* is defined by using the Paradigms module function taking the corresponding word stem as an argument and doing the necessary type casting operations in a very similar way in all five languages:

```
 -- English:
 PainKiller = cnNonhuman "painkiller" ;
 -- French:
 PainKiller = mkCN (nReg "calmant" masculine) ;
 -- Italian:
 PainKiller = mkCN (nSale "calmante" masculine) ;
 -- Swedish:
 PainKiller = mkCN (nIngenBöjning "smärtstillande") ;
 -- Russian:
 PainKiller = mkCN obezbolivauchee ;
```

Porting of the lexical functions is also relatively easy. We just need to provide the stems for rules from the Paradigms module.

In the remaining three functions we see bigger differences. For example, the idiomatic expression "I have a cold"in Swedish and Russian is formed by adjective predication not with transitive verb like in English, French and Italian. Therefore, the different functions `PosA` and `PosTV` are used. `tvHave`, `tvAvoir` and `tvAvere` denote the transitive verb *have* in English, French and Italian. Expressions in the parentheses are needed for the type compatibility described in subsection 3.3.1:

```
 -- English:
 CatchCold = PosTV tvHave (DetNP aDet (cnNoHum "cold"));
 -- French:
 CatchCold = PosTV tvAvoir (IndefOneNP
                    (mkCNomReg "rhume" masculine));
 -- Italian:
 CatchCold = PosTV (tvAvere)(IndefOneNP
                    (mkCN (nSale "raffreddore" masculine))) ;
 -- Swedish:
 CatchCold = PosA (adjGrund "förkyld") ;
 -- Russian:
 CatchCold = PosA prostuzhen ;
```

The phrase *I need a painkiller* is a transitive verb predication together with complementization rule defined in subsection 3.2.4 in English and Swedish. In French and Italian , however, we need to use the idiomatic expressions *avoir besoin* and `averBisogno` correspondingly and, therefore, the more basic rule `PredVP` is used. In Russian the same meaning is expressed by using adjective predication. A special predicate is defined in the Russian syntax module and, therefore, the arguments `patient` and `medicine` are omitted in the definition below:

```
-- English:
NeedMedicine patient medicine = predV2 (mkTransVerbDir
             (regVerbP3 "need")) patient (DetNP aDet medicine);
-- Swedish:
NeedMedicine patient medicine = predV2 (mkDirectVerb verbBehova)
                              patient (DetNP nullDet medicine) ;
-- French:
NeedMedicine patient medicine = PredVP
                              patient (avoirBesoin medicine) ;
-- Italian:
NeedMedicine patient medicine = PredVP
                              patient (averBisogno medicine) ;
-- Russian:
NeedMedicine                  = predNeedShortAdjective True;
```

Notice, that the medicine argument is used with an indefinite article in English version, with empty article in Swedish, French and Italian. Russian does not have any articles although the category `Determiner` is present for the sake of consistency with the language-independent API.

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

### 3.3.3 Arithmetic grammar usage example

Finally, we give an example of using the whole arithmetic grammar (see also section 2.1.6). Fig. 3.2 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 (see section 2.1).

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.

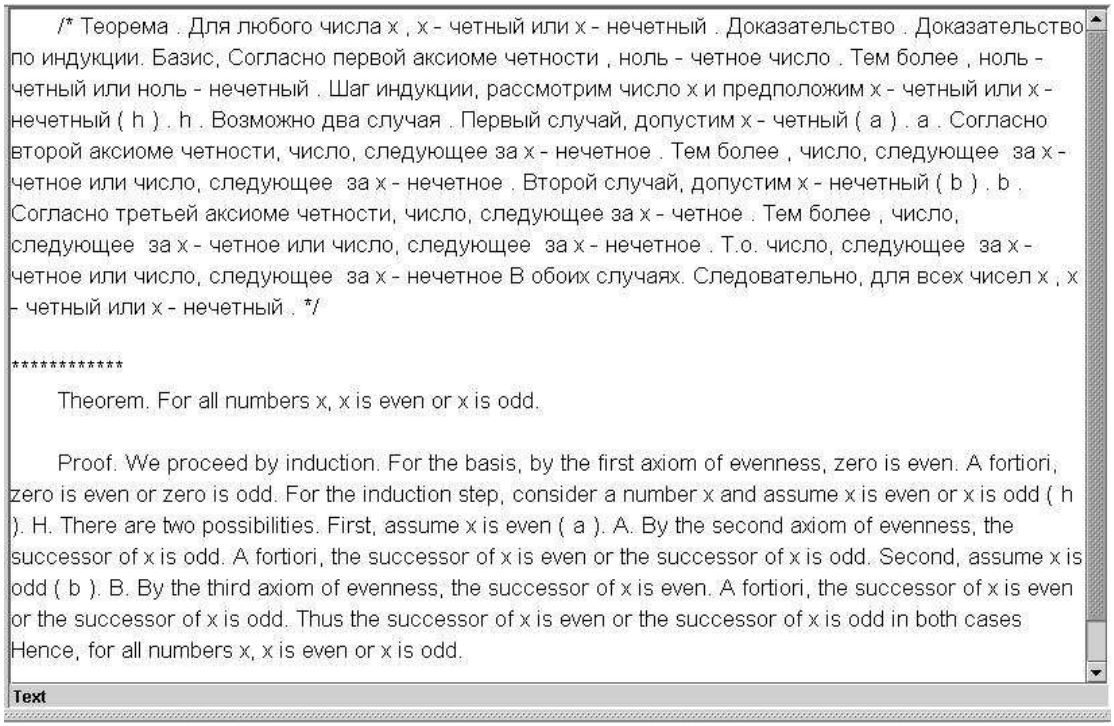


Figure 3.2: Example of a theorem proof constructed using arithmetic grammars in Russian and English.

# Chapter 4

# Functional parsing for biosequence analysis

In this chapter we try to evaluate the prospects of applying standard *functional parsing* techniques to the *biosequence analysis* problem. It is based on the report [J.K02] on three-month joint project with Laboratory of Informatics at École Polytechnique, supported by a grant from École Polytechnique foundation.

Biosequence analysis is one of the most challenging problems in the relatively new and rather popular field of bioinformatics. The ongoing research at Ecole Polytechnique (France) takes the grammar approach to the problem of detecting multi-level protein structure [J.W00, Ber96]. The first order protein structure is just a sequence of aminoacids. Considering each type of aminoacid as an alphabet letter, context-free grammars are used for describing the higher level structures. By parsing of the aminoacid chain we get the higher level structures of the corresponding protein. In the general case, however, the protein structure cannot be described by context-free grammars.

Functional languages provide a number of elegant standard methods for building parsers for both context free and context-dependent grammars. First option is to use *parser generators* like Happy [Mar02] with predefined grammar constructors. In the next part we will illustrate this approach by Grammatical Framework (GF), which can be classified as a parser generator. Second option is to use *parser combinator* libraries, which offer the full power of the functional language at user's disposition. Parsing library written in Haskell is chosen as an example of the later technique.

## 4.1 Sample biosequence grammar

The aminoacid sequence describes the so called primary structure of a protein. According to bioscience theory the primary structure determines the secondary structure (consisting of sequence of aminoacid groups of different types), which,

in turn, determines the 3D structure of protein, which, in turn, affects the properties of a protein. Unfortunately, the exact laws of how to form a higher order structure from a lower one are unknown, therefore, we can only speak about protein structure *prediction*.

Bioscience in general is highly interested in 3D structure predictions, because they can provide useful hints on various bio-mechanisms: "double spiral"shape of DNA (Deoxyribonucleic acid) is an instance in point.

It's much easier to find the primary than the 3D structure experimentally. Therefore, the purpose of biosequence analysis is to predict the higher structures giving the primary structure.

The simplest biosequence grammar model corresponds to the language that comprise 0 and 1 as the only "words"(corresponds to hydrophobic and hydrophilic aminoacids respectively). "Statements"represent the secondary structure of a protein. The "statements"(around 100 word length) are all possible sequences of the form XYXY...XYX, where

- X - an arbitrary sequence of 0 and 1 usually short (less than 5 digits) or even empty. This is so called "noise".
- Y - is either $\alpha^n$ or $\beta^m$ or $\gamma^k$, where
  - both $\alpha$ and $\gamma$ are 5-digit sequences of the known form (namely the binary representation of numbers: 9, 19, 6, 12, 25, 18 or 4)
  - $\beta^m$ is 4-digit sequence of the known form (namely the binary representation of numbers 10 or 5)

There are also some restrictions:

- $m$ should be the same for every Y that contains $\beta$ occurences,
- $n$ should be the same for every Y that contains $\alpha$ occurences (so calles "coupled"$\alpha$-sequences),
- $k$ could vary among Ys, that contain $\gamma$ occurences.

From the description one can see that the grammar is highly ambiguous, but this is just half the trouble. The main purpose is to find an "optimal"parsing of 0-1-sequence into $\alpha - \beta - \gamma$ representation. The goal function could look like:

- first, maximize [n$\times$ the number of $\alpha^n$ sequences],
- second, maximize [m$\times$ the number of $\beta^m$ sequences]
- finally, maximize $\sum_j k_i$

According to bioscience theory the optimal parsing represents the real form of a protein. In fact, finding of an appropriate goal function is itself a big problem in bioscience. We are satisfied with the choice of the goal function as soon as we get biologically reasonable results.

## 4.2 GF as a functional parser generator

Grammatical Framework (GF)[Ran03] is a multilingual translation system of INTERLINGUA type. It implements the language for expressing grammars using formalism based on type theory [Ranar]. GF accepts context-dependent grammars due to implemented Context Sensitive Parsing (CSP).

CSP consists of Context Free Parsing and independent filtering or postprocessing. The latter can be compared to so called Context Free Rewriting Systems (CFRS) although CFRS allows only permutation of the arguments in the otherwise context-free rules while GF also supports more complicated dependencies like reduplication and suppression.

GF has a rich high-level grammar description language, that allows to express various features specific to natural languages in a straightforward manner. However, when it comes to highly ambiguous bioseqience grammars the strong need for optimizing calculations cannot be satisfied by GF's limited constructor set.

Lets consider a very simple grammar for $\beta$ sequences:

S → noise B | B noise | noise B noise
B → $B_1$ | $B_0$
$B_0$ → 0 $B_1$
$B_1$ → 1 $B_0$
$B_0$ → 0101
$B_1$ → 1010
noise → noise noise | 0 | 1

Biosequence analysis could be regarded as translation from aminoacid chain into combination of higher order protein structures. To implement the translator we need to describe an abstract syntax, which contains the general description of the grammar, input and output concrete syntax, which linearize the abstract part. Input syntax deals with aminoacid chain. Output syntax generates the position of higher order $\beta$-structure.

```
-- abstract part ---------------------------------------

cat S , B , HFil, HFob, BHFil, BHFob, V, Rec ;


fun Zero :  HFob;
 One :  HFil;
 VZero :  HFob -> V;
 VOne :  HFil -> V;
 Ten:  HFob -> HFil -> HFob -> HFil -> HFob -> BHFob;
 Twenty:  HFil -> HFob -> HFil -> HFob -> HFil -> BHFil;
 Fob :  BHFil -> BHFob ;
```

```
 Fil :  BHFob -> BHFil;
 BFil :  BHFil -> B ;
 BFob :  BHFob -> B ;
 First :  B -> Rec;
 Second :  B -> B -> Rec;
 Third :  B -> B -> B -> Rec;
 ListLeft :  V -> Rec -> S ;
 ListRight :  V -> Rec -> S ;
 ListBoth :  V -> V -> Rec -> S ;
 ListNone :  Rec -> S ;
 Noise :  V -> V -> V;

-- input concrete part ----------------------------------------

include b.Abs.gf ;
lin Zero = { s = "0"} ;
 One = { s = "1"} ;
 Ten a b c d e = { s = a.s ++ b.s ++ c.s++ d.s ++ e.s } ;
 Twenty a b c d e = { s = a.s++ b.s++ c.s++ d.s++ e.s } ;
 Fob x = { s = "0"++ x.s } ;
 Fil x = { s = "1"++ x.s } ;
 VZero x = { s = x.s } ;
 VOne x = { s = x.s } ;
 BFil x = { s = x.s } ;
 BFob x = { s = x.s } ;
 First a = { s1= a.s } ;
 Second x a = { s1 = a.s ++ x.s } ;
 Third k x a = { s1 = k.s ++ a.s ++ x.s } ;
 ListRight x a = { s = a.s1 ++ x.s } ;
 ListLeft x a = { s = x.s ++ a.s1 } ;
 ListBoth x y a = {s = x.s ++ a.s1 ++ y.s};
 ListNone a = {s = a.s1 };
 Noise x y = { s = x.s ++ y.s } ;

-- output concrete part ---------

include b.Abs.gf ;
lin Zero = { s = "0"} ;
 One = { s = "1"} ;
 Ten a b c d e = { s = "BBBBB"} ;
 Twenty a b c d e = { s = "BBBBB"} ;
 Fob x = { s = "B"++ x.s } ;
 Fil x = { s = "B"++ x.s } ;
```

```
VZero x = { s = x.s } ;
VOne x = { s = x.s } ;
BFil x = { s = x.s } ;
BFob x = { s = x.s } ;
First a = { s1= a.s } ;
Second x a = { s1 = a.s ++ x.s } ;
Third k x a = { s1 = k.s ++ a.s ++ x.s } ;
ListRight x a = { s = a.s1 ++ x.s } ;
ListLeft x a = { s = x.s ++ a.s1 } ;
ListBoth x y a = {s = x.s ++ a.s1 ++ y.s};
ListNone a = {s = a.s1 };
Noise x y = { s = x.s ++ y.s } ;
```

Let us generate random aminoacid chain that corresponds to the defined grammar, then linearize it and finally translate aminoacid chain into sequence containing $\beta$-structure:

```
> trace gr 1 S | l| t -td2 Bin Bet S

ListRight(VOne One)(First(BFil(Fil(Fob(Fil
 (Ten Zero One Zero One Zero))))))
101010101
 1BBBBB101
 1BBBBBB01
 1BBBBBBB1
 1BBBBBBBB
 10BBBBB01
 10BBBBBB1
 10BBBBBBB
 101BBBBB1
 101BBBBBB
 1010BBBBB
 BBBBB0101
 BBBBBB101
 BBBBBBB01
 BBBBBBBB1
 BBBBBBBBB
```

Since the grammar is ambiguous we get many parsing combinations. Within GF language there is no mechanism to evaluate, which of them is the best and therefore it's impossible to filter the results to reduce the search space. Without extra calculations the search space quickly becomes exponential and therefore fails to produce acceptable results for more ambiguous grammars.

## 4.3 Parser combinators

Parser combinators libraries [CM95] are more appealing than parser generators, since we can use the full power of functional language to extend the capabilities for expressing grammars.

In the example below we can, for instance, calculate the energy function for each parsing result and consequently choose one according to this criterion. Such arithmetic operations were impossible in GF.

Monadic approach [GM96] specific to Haskell allows to write very general parsers rather elegantly. Nevertheless, in order to cope with exponential growth we need not only adjust the existing set of parser combinators, but also rewrite it.

There is a very simple example of biosequence grammar, which uses a little bit more complex model of aminoacids:

$one \rightarrow noise\ B\ noise$
$B \rightarrow B_0 \mid B_1$
$B_0 \rightarrow hFob\ B_1$
$B_1 \rightarrow hFil\ B_0$
$B_0 \rightarrow hFob\ hFil\ hFob\ hFil\ hFob$
$B_1 \rightarrow hFil\ hFob\ hFil\ hFob\ hFil$
$hFil \rightarrow a|r|n|c|g|h|i|l|m|f|s|w|y|v$
$hFob \rightarrow a|n|d|q|e|g|h|k|p|s|t$
$noise \rightarrow hFob \mid hFil \mid noise^*$

Unlike the example of the previous section now the alphabet is not restricted to 1 and 0. Namely, the hydrophobic/hydrophilic property is not just true or false. Instead each letter represents aminoacid and has a hydrophobicity coefficient. If this coefficient is close to 0 the corresponding aminoacid could be interpreted as both hydrophilic and hydrophobic aminoacid. Besides, the minimum length of $\beta$-structure is 5 instead of 4. The parser recognizes $\beta$-sequence in the input aminoacid chain. Here is the code in Haskell:

```
module Beta where
import List import ParseLib

fone ::  (Char, Float) -> Parser (Char, Float)
fone (c, n) = do  char c; return (c,n)

hFob ::  Parser (Char, Float)
hFob = foldl (+++) mzero (map fone foldl
 [('a', 0.22),
 ('n', -0.46),
 ('d', -3.08),
```

```
 ('q', -2.81),
 ('e', -1.81),
 ('g', 0),
 ('h', 0.46),
 ('k', -3.04),
 ('p', -2.23),
 ('t', -1.90),
 ('s', -0.45)])

hFil ::  Parser (Char, Float)
hFil = foldl (+++) mzero (map fone foldl
 [('a', 0.22),
 ('r', 1.42),
 ('n', -0.46),
 ('c', 4.07),
 ('g', 0),
 ('h', 0.46),
 ('i', 4.77),
 ('l', 5.66),
 ('m', 4.23),
 ('f', 4.44),
 ('s', -0.45),
 ('w', 1.04),
 ('y', 3.23),
 ('v', 4.67)])

beta51 ::  Parser (String, Float)
beta51 = do {a1<-hFil;a2<-hFob;a3<-hFil;
 a4<- hFob;a5 <- hFil;}
 return(("BBBBB"),(snd(a1)-snd(a2)+snd(a3)-snd(a4)+snd(a5)))}

beta50 ::  Parser (String, Float)
beta50 = do {a1<-hFob;a2 <-hFil; a3<-hFob;
 a4 <-hFil; a5 <- hFob;}
 return (("BBBBB"),(snd(a1)-snd(a2)+snd(a3)-snd(a4)+snd(a5)))}

beta1 ::  Parser (String, Float)
beta1 = do{a1<-hFil;a2<-beta0;}
 return(('B':fst(a2)),snd(a1)-snd(a2))}+++ beta51

beta0 ::  Parser (String, Float)
beta0 = do { a1 <- hFob; a2 <- beta1;}
 return (('B':fst(a2)),snd(a1)-snd(a2)) } +++ beta50

beta ::  Parser (String, Float)
```

```
beta = beta1 +++ beta0

many0 ::  Parser a -> Parser [a]
many0 p = force ( many01 p `mplus` (return []))

many01 ::  Parser a -> Parser [a]
many01 p = do { a <- p; as <- many0 p; return (a:as) }

noise1 ::  Parser (String, Float)
noise1 = do{ a <- hFob; return ([fst(a)],0) } +++
 do { a <- hFil; return ([fst(a)],0) }

helpFun ::  (String, Float) -> (String, Float) ->
(String, Float)
helpFun = \ x y -> (fst(x)++fst(y), 0)

noise ::  Parser (String, Float)
noise = do {xs <- many0 noise1;
return(foldl helpFun ("",0) xs)}

one ::  Parser (String, Float)
one = do { a1 <- noise; a2 <- beta; a3 <- noise;}
 return (fst(a1)++(fst(a2)++fst(a3)),snd(a2))

compareBy ::  a -> a -> (a -> Float) -> Ordering
compareBy x y f = if (f(x)>f(y)) then LT else
 if (f(y)>f(x)) then GT else EQ

better::((String,Float),String)->((String,Float),String)->
Ordering
better x y = compareBy x y (abs.snd.fst)

rightOrder::[((String,Float),String)]->
[((String,Float),String)]
rightOrder l = sortBy better l

final ::  String -> [((String, Float),String)]
final s = genericTake 1 (rightOrder (papply one s))
```

Parser returns the detected structures together with energy value. Because of the ambiguity there are several parsing results:

```
Beta> papply one "aarncdqek"

 [(("aBBBBBqek",-8.81),""),(("aBBBBBqe",-8.81),"k"),
 (("aBBBBBq",-8.81),"ek"),(("aBBBBB",-8.81),"qek"),
 (("BBBBBBqek",9.03),""),(("BBBBBBqe",9.03),"k"),
```

```
(("BBBBBBq",9.03),"ek"),(("BBBBBB",9.03),"qek")]
```

According to chosen optimization criteria we are interested in the result with the highest absolute energy value:

```
Beta> final "aarncdqek"

[(("BBBBBBqek",9.03),"")]
```

The problem with this example is that the parser evaluates all the results and then chooses the best. It leads to exponential growth. To deal with the problem dynamic programming is usually used. It allows to cut non-optimal branches as early as possible and avoid repetition of the same calculations.

To go further we need a deeper analysis of the domain area. Namely, we need to be sure that the goal function we use doesn't cut the results we might be interested in. The design of goal functions for biosequence analysis is a complex problem. There exists a number of theories to predict the secondary structure of proteins given the primary structure. The goal function used above originates in physics: each aminoacid is assigned an energy value, which characterizes its hydrophobicity. The optimal protein secondary structure is the one that maximizes the total energy value.

## 4.4 Conclusion

Using both standard ways of functional parsing we can write clear and easy to read grammars for biosequences. But due to ambiguity of grammars the performance remains the main obstacle to successful application.

The power of functional programming is based on strong typing. Unfortunately, we are not able to take much advantage of this property due to the very ambiguous nature of biosequence grammars.

One way to handle the problem is to use controlled dynamic parsing where we can interfere and cut some parsing branches as well as keep track of calculated results to avoid doing the same work twice. The main problem is to find a goal function that allows us to make cutting decisions as early as possible and that is still able to produce satisfactory results. The choice of goal function is a well-known problem in bioinformatics irrelative to programming issues. Dynamic programming and other approaches are used in bioinformatics software [Bio03] usually implemented in imperative languages like C.

GF can be used for describing biosequence grammars, but the parser routinely created is too inefficient. A possible approach can be to use GF for description, and work with specialized parsers.

# Chapter 5

# Conclusion

## 5.1 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. We do not know about other projects trying to combine bioinformatics and functional programming, so this part will be left without comparison to related work.

### 5.1.1 Multilingual authoring

The idea of developing user interface in Java programming language comes from the CtCoq system [Y. 99].

GF applies formal language technique to natural languages. Syntax editing procedure is originated from proof editors like Alf [MN94] 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.

Multilingual authoring approach is similar to the one in the WYSIWYM tool [vDP00, PS98]. 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 the WYSIWYM corresponds to the construction of an abstract syntax tree in GF. The refinement entities called anchors in the 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 the WYSIWYM corresponds to a grammar (Abstract part) in GF.

Even the architecture of one of the WYSISYM implementations DRAFTER-II [PSE98] 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 GUI is written in CLIM (Common Lisp Interface Manager).

However, unlike the 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. WYSIWYM, therefore, corresponds to a Gramlet specialized for one grammar. 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 the WYSIWYM and GF are quite different in nature. The GF model is more semantic oriented, while the WYSIWYM is focused on syntactic information. Syntax is more developed and even stylistic issues are addressed in the recent WYSIWYM applications [PSH03]. 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 the WYSIWYM. 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. This is considered useful for multilingual authoring applications because typing can speed up the tedious syntax editing procedure.

GF was one of the sources of inspiration for an XML-based multilingual document authoring application for pharmaceutical domain developed at XRCE [BDL00, DLR00]. The grammar formalism used in this system is called Interaction Grammars (IG). 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 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. It is based on the Definite Clause Grammars - a unification-based extension of context-free grammars, which has a build-in implementation in Prolog.

### 5.1.2 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 [MDP+00].

Like the resource grammars in GF the CLE grammars aimed to be domain-independent, however they were trained upon and built with 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 trasfer 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 the [MDP+00] 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 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. As for GF:

- Numerals are not represented as a grammatical category. Cardinal numerals, however, are represented in an application grammar.
- Gerund is not yet covered and past participle is only treated as adjective.
- No complex verb phrases (with more than one verb part (auxiliary, modal, infinitive)) and also no verb phrase conjunction.

The same phenomena are not treated in the same way. Prepositional phrases are handled by TV and Adverb types in GF.

Verb phrase discontinuous constituents are handled by combining the record fields (see 3.2.3) 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. 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-inversed word order we would need to use different rules for V and S. The forth 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 handling questions:

```
questVerbPhrase' : Bool -> NounPhrase -> VerbPhrase -> Question =
  \adv,john,walk -> {s = table {

      DirQ   => if_then_else Str walk.isAux

          (indicVerb (verbOfPhrase walk) john.p john.n ++
          john.s ! NomP ++ walk.s2 ! john.n)

          (indicVerb verbP3Do john.p john.n ++
          john.s ! NomP ++ walk.s ! InfImp ++ walk.s2 ! john.n) ;
```

```
      IndirQ => if_then_else Str adv

          []

          (variants {"if" ; "whether"}) ++
          (predVerbPhrase john walk).s
      }
    } ;
```

For example, in the rule above in the lines four and five we can see that the verb phrase denoted by `walk` [*be a man*] is used twice: in the first word `indicVerb (verbOfPhrase walk) john.p john.n` [*are*] and in the third word `walk.s2 ! john.n` [*men*], i.e. the verb phrase is discontinuous: different parts of the verb phrase are used in different parts of the sentence. 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 `john`). In GF this is done by selecting (using the exclamation mark `!`) the plural form `walk.s2 ! john.n` from the table `walk.s2`, where `john.n` is 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 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 3.2.2) while several independent rules are needed 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 restriction, i.e. all the inflection forms of a word are described in one rule (function).

Such differences 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.

## 5.2 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 several senses:

### Multi-lingual

The GF grammar language was used for building grammars in Russian.

The main result in this area is Russian resource grammar library adapted from the similar libraries for other languages. Such libraries allows 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. (providing that they are properly used, i.e. no type errors and the right rules are chosen).

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 handling the Cyrillic fonts display.

### Multi-platform

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. Gramlets functionality is restricted to syntax editing and linearizing, but no parsing or dependent types like in GF written in Haskell.

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 now 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 tool [HJR02], 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. 5.1 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.

#### Multi-domain

We also tried applying GF to a non-linguistic domain, namely bio-informatics. Functional parsing in general and GF in particular were used for protein grammar description. What we wanted was to describe the protein structure as a grammar and try to use parsing for determining the structure of proteins. The problems we encountered were related to the ambiguity of the protein grammar, that turned out to be overwhelming for GF designed for natural languages. The results were not too encouraging, but we certainly have gained some insights on GF strengths and limitations during the attempts.

## 5.3 Future work

So far the GF grammars has been developed interactively according to the build-try-improve circular model. We are aiming at creating an Integrated Development Environment (IDE) for GF language that contains tools for users on different levels.

The main direction for future work is to treat the GF grammar formalism itself as a GF grammar, which means to write a grammar editor that will provide a convenient environment for writing GF grammars. Some command line tools are provided by GF for grammar processing, but no integrated graphical development environment (IDE) for creating grammars exists for the moment. For example, now written grammars can be type checked together with compilation in a batch mode. Type checking ensures that no run-time type errors occur, which is the whole point of using typing in programming languages. However, it would be even better if a rule could be type checked immediately during the writing process. Rule construction in such an IDE would become similar to constant definition construction in a proof editor. Unlike the present Java GUI syntax editor such a

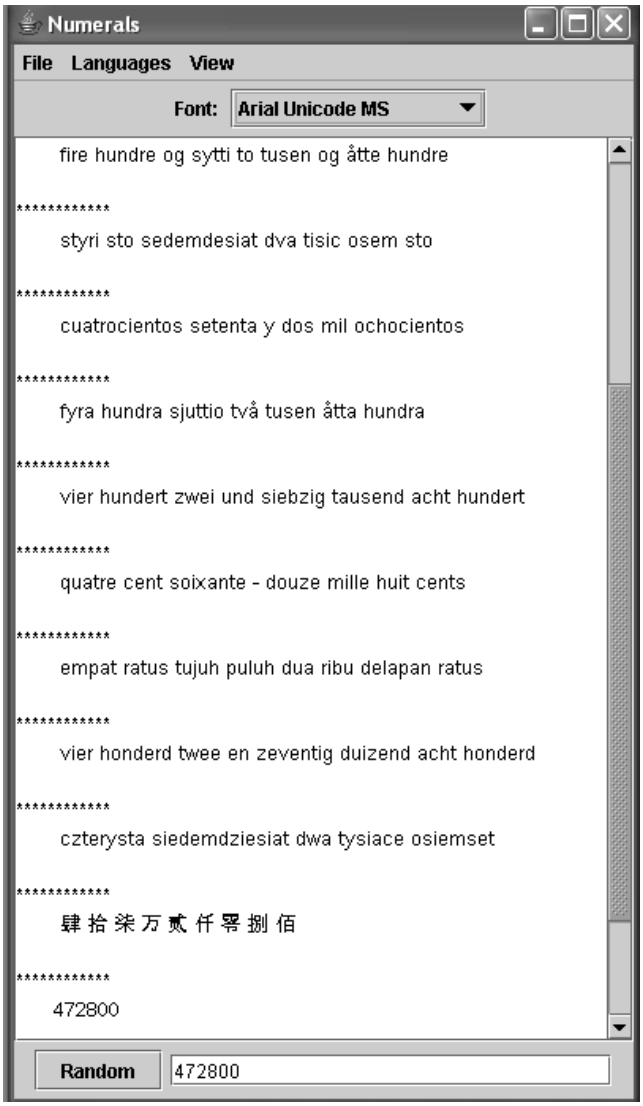


Figure 5.1: Numerals GUI displays numbers in more than 20 languages. Adapted from the Java Syntax Editor GUI without disturbing the GF core system.

tool will belong to a higher — grammarian — user level. We would like to explore the possibility to introduce more visual tools in the editing process. Ability to perform function search in the resource library will also be of use for grammar writers.

We think that work on the GF supporting tool functionality will provide room for more theoretical investigations compared to the work presented. For instance, in case of writing a GF grammar editor this is caused by less previous work to build on and by the non-triviality of the implementation task of some desirable features.

Once written any software needs to be maintained. This means that the work on the Syntax Editing and the resource libraries will continue.

Among the features to be implemented in Java GUI syntax editor is syntax editing of text in the linearizations area. The possibility to edit several objects simultaneously can be of some use, especially in the light of stronger visualization of the editing process. For example, editing objects in different windows and then combining them by cut&paste operation sounds like a useful feature. The present Java GUI module as well as the XML communication protocol can be a subject to change. The Gramlets performance will probably gain from a different implementation. After all, despite the strong shift to computational linguistics field building an NLP system remains, in the first place, a matter of software engineering when it comes to working applications.

The resource library and its structure can be further extended and modified. The main purposes of the resource library are language coverage and high-level API to make it relatively easy-to-use. Writing hand-coded grammars requires considerable investment of effort and has been strongly challenged by the surface processing using statistical methods. The GF system argues for high-quality translation in a limited domain. The resource grammar library motivation is to lower the grammar development cost without compromising the expressive power of the GF language formalism.

# Appendix A

# 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.

## A.1 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.

## A.2 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.

## A.3 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.

## A.4 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).

- *Menus* – 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).

## A.5 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.

## A.6 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 [Ran03].
- *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.

# Appendix B

# Automatically generated test examples

Automatically generated test examples of using the resource grammar library functions (in English and in Russian) are intended for proof-reading and also reflect the coverage of the resource library. Similar examples are generated in other languages (French, German and Italian). In some examples, several linearizations of the same syntax tree (paraphrases) are possible like in the second example - four variants are present. When working in the Syntax Editor only the first variant is shown.

**Example 1.**

He says that you run well.
He says you run well.

**Example 2.**

She does not send the big, small or either old or young car.
She does not send the big, small or either old or young car.
She doesn't send the big, small or either old or young car.
She doesn't send the big, small or either old or young car.

Figure B.1: Examples from English resource library.

**Example 3.**

If John does not walk always, we switch on a light in every old house.
If John does not walk always, we switch a light on in every old house.
If John doesn't walk always, we switch on a light in every old house.
If John doesn't walk always, we switch a light on in every old house.
We switch on a light in every old house, if John does not walk always.
We switch on a light in every old house, if John doesn't walk always.
We switch a light on in every old house, if John does not walk always.
We switch a light on in every old house, if John doesn't walk always.

**Example 4.**

Either they are younger than you or I am a mother of the biggest man.

**Example 5.**

The mothers always of yours prove that you are small.
The mothers always of yours prove you are small.
Your mothers always prove that you are small.
Your mothers always prove you are small.

**Example 6.**

Men such that the uncles of theirs love neither the cars nor the houses are not old.
Men such that the uncles of theirs love neither the cars nor the houses aren't old.
Men such that the uncles of theirs love neither the cars nor the houses are not old.
Men such that the uncles of theirs love neither the cars nor the houses aren't old.
Men such that their uncles love neither the cars nor the houses are not old.
Men such that their uncles love neither the cars nor the houses aren't old.
Men such that their uncles love neither the cars nor the houses are not old.
Men such that their uncles love neither the cars nor the houses aren't old.

Figure B.2: Examples from English resource library.

**Example 7.**

When a man that does not run walks, who runs?
When a man that doesn't run walks, who runs?
When a man who does not run walks, who runs?
When a man who doesn't run walks, who runs?
Who runs, when a man that does not run walks?
Who runs, when a man that doesn't run walks?
Who runs, when a man who does not run walks?
Who runs, when a man who doesn't run walks?

**Example 8.**

The car is not big, the house is small and she is old.
The car isn't big, the house is small and she is old.

**Example 9.**

When you are young, walk!
Walk, when you are young!

**Example 10.**

The uncles of which women do I wait for?
For the uncles of which women do I wait?

**Example 11.**

Why do they love Mary?
Why do they love Mary?

**Example 12.**

Are we men?

**Example 13.**

Yes.

Figure B.3: Examples from English resource library.

**Example 14.**

Most cars.

**Example 15.**

Old houses.

**Example 16.**

How?

**Example 17.**

She is a woman the mother of whom he waits for.
She is a woman for the mother of whom he waits.

**Example 18.**

Mary, John or either you or I am young.

**Example 19.**

I love old houses and young women.
I love old houses and young women.

**Example 20.**

Cars parked near the Russian house.

Figure B.4: Examples from English resource library.

**Example 1.**

он говорит, что ты хорошо бегаешь.
он говорит, что ты хорошо бегаешь.
он говорит, что ты хорошо бегаешь.
он говорит, что ты хорошо бегаешь.

**Example 2.**

она не отправляет большую, маленькую или либо старую, либо молодую машину.

**Example 3.**

если Иван не всегда гуляет, мы в каждом старом доме включаем свет.
если Иван не всегда гуляет, мы в каждом старом доме включаем свет.
мы в каждом старом доме включаем свет, если Иван не всегда гуляет.
мы в каждом старом доме включаем свет, если Иван не всегда гуляет.

**Example 4.**

либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.
либо они - моложе вас, либо я - мама самого большого мужчины.

**Example 5.**

ваши мамы всегда доказывают, что вы - маленькие.
ваши мамы всегда доказывают, что вы - маленькие.

**Example 6.**

мужчины, такие что их дяди любят ни машины, ни дома - не старые.

Figure B.5: Examples from Russian resource library.

**Example 7.**

когда мужчина, который не бегает гуляет, кто бегает?
кто бегает, когда мужчина, который не бегает гуляет?

**Example 8.**

машина - не большая, дом - маленький и она - старая.

**Example 9.**

когда вы - молодые, гуляйте!
когда вы - молодые, гуляйте!
гуляйте, когда вы - молодые!
гуляйте, когда вы - молодые!

**Example 10.**

дядей каких женщин я жду?
дядей каких женщин я жду?
дядей каких женщин я жду?
дядей каких женщин я жду?
каких женщин дядей я жду?
каких женщин дядей я жду?
каких женщин дядей я жду?
каких женщин дядей я жду?

**Example 11.**

почему они любят Машу?
почему они любят Машу?

**Example 12.**

мы - мужчины?
мы - мужчины?

Figure B.6: Examples from Russian resource library.

**Example 13.**

да.

**Example 14.**

большинство машин.

**Example 15.**

старые дома.

**Example 16.**

как?

**Example 17.**

она - женщина, маму которой он ждет.

**Example 18.**

Маша, Иван или либо вы, либо я - молодые.

**Example 19.**

я люблю старые дома и молодых женщин.
я люблю старые дома и молодых женщин.
я люблю старые дома и молодых женщин.
я люблю старые дома и молодых женщин.

**Example 20.**

запаркованные около русского дома машины.

Figure B.7: Examples from Russian resource library.

# Appendix C

# Types module

This is a resource module for Russian morphology, defining the morphological parameters and word classes of Russian. It is aimed to be complete w.r.t. the description of word forms. However, it does not include those parameters that are not needed for analysing individual words: such parameters are defined in syntax modules.

```
include ../prelude/prelude.gf ;
```

## C.1 Enumerated parameter types

These types are the ones found in school grammars. Their parameter values are atomic.

```
param
  Gender     = Masc | Fem | Neut ;
  Number     = Sg | Pl ;
  Case       = Nom | Gen | Dat | Acc | Inst | Prepos ;
  Voice      = Act | Pass ;
  Aspect     = Imperfective | Perfective ;
  Tense      = Present | Past ;
  Degree     = Pos | Comp | Super ;
  Person     = P1 | P2 | P3 ;
  AfterPrep  = Yes | No ;
  Possessive = NonPoss | Poss GenNum ;
  Animacy    = Animate | Inanimate ;
```

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

## C.2 Word classes and parameter types

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).

### C.2.1 Nouns

Common 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 ;
```

Substantives moreover have an inherent gender.

```
oper
   CommNoun : Type = {s : SubstForm => Str ; g : Gender ;
                      anim : Animacy } ;
   numSF: SubstForm -> Number = \sf -> case sf of
    {
      SF Sg _ => Sg ;
      _       => Pl
    } ;

   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
    } ;
```

### C.2.2 Pronouns

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

param  PronForm = PF Case AfterPrep Possessive;
```

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

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

The following coercion is useful:

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

  -- the best we can do for я, ты

    } ;

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 ;
```

### C.2.3 Adjectives

Adjectives is a very complex class. The major division is between the comparison degrees.

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

Declination forms depend on Case, Animacy , Gender: *новые дома - новых домов* (new houses - new houses'), Animacy plays role only in the Accusative case: *я люблю новы-е дома – я люблю новы-х мужчин* (I love new houses - I love new men); and on Number: *новый дом - новые дома* (a new house - new houses). The plural never makes a gender distinction.

```
  GenNum = ASg Gender | APl ;

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

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

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

oper caseAF: AdjForm -> Case = \af ->
   case af of { 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*) - *наивысший* (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 - болвной* (long, predicative) vs. *он - болен* (short, predicative).

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

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} ;
```

### C.2.4 Verbs

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 *ли*. 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 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 - с-ломать - по-ломать* (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: http://starling.rinet.ru/morph.htm Note: Of course, the whole inflection table has many redundancies in a 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). 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 VTense | VIMP Number Person | VINF |
             VSUB GenNum ;
  VTense   = VPresent Number Person | VPast GenNum |
             VFuture Number Person ;
```

For writing an application grammar one usually doesn't need the whole inflection table, since each verb is used in a particular context that determines some of the parameters (Tense and Voice while Aspect is fixed from the beginning) for certain usage. So we define the *Verb* type, that have these parameters fixed. The conjugation parameters left (Gender, Number, Person) are combined in the *VF* type:

```
param VF = VFin GenNum Person | VImper Number Person |
           VInf | VSubj GenNum;

oper
  Verb : Type = {s : VF => Str ; t: Tense ; a : Aspect ;
         v: Voice} ;

  extVerb : Verbum -> Voice -> Tense -> Verb = \aller, vox, t ->
```

```
  { s = table {
     VFin gn p => case t of {
     Present => aller.s ! VFORM vox (VIND (VPresent
                          (numGNum gn) p)) ;
     Past =>   aller.s ! VFORM vox (VIND (VPast gn))
     } ;
     VImper n p => aller.s ! VFORM vox (VIMP n p) ;
     VInf => aller.s ! VFORM vox VINF ;
     VSubj gn => aller.s ! VFORM vox (VSUB gn)
     }; t = t ; a = aller.asp ; v = vox } ;
```

### C.2.5 Other open classes

Proper names and adverbs are the remaining open classes.

```
oper
  PNm    : Type = {s : Case => Str ; g : Gender} ;
```

Adverbials are not inflected (we ignore comparison, and treat compared adverbials as separate expressions; this could be done another way).

```
  Adverb : Type = SS ;
```

### C.2.6 Closed classes

The rest of the Russian word classes are closed, i.e. not extensible by new lexical entries. Thus we don't have to know how to build them, but only how to use them, i.e. which parameters they have.

### C.2.7 Relative pronouns

Relative pronouns are inflected in gender, number, and case just like adjectives.

```
  RelPron : Type = {s : GenNum => Case => Animacy => Str} ;
```

### C.2.8 Prepositions are just strings.

```
  Preposition = Str ;
```

# Appendix D

# A Small Russian Resource Syntax

This resource grammar contains definitions needed to construct indicative, interrogative, and imperative sentences in Russian.

The following files are presupposed:

```
include
  morpho.RusU.gf ;   -- ad hoc morpho rules
  ../prelude/prelude.gf ;      -- language independent prelude
  types.RusU.gf ;    -- used in functional morphology as well
  coordination.gf ; -- language-independent coordination package
```

## D.1 Common Nouns

### D.1.1 Common noun phrases

Complex common nouns (`Comm'NounPhrase`) have in principle the same parameters as simple ones.

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

  noun2CommNounPhrase : CommNoun -> CommNounPhrase =
  \sb ->
    {s = \\n,c => sb.s ! SF n c ;
     g = sb.g ;
     anim = sb.anim
    } ;
```

```
commNounPhrase2CommNoun : CommNounPhrase -> CommNoun =
\sb ->
  {s = \\sf => sb.s ! (numSF sf) ! (caseSF sf) ;
   g = sb.g ;
   anim = sb.anim
  } ;

n2n = noun2CommNounPhrase;
n2n2 = commNounPhrase2CommNoun ;
```

## D.2 Noun Phrases

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

  -- A function specific for Russian for setting the gender for
  -- personal pronouns in first and second person, singular :
  setNPGender : Gender -> NounPhrase -> NounPhrase =
  \gen, pronI ->
    { s = pronI.s ; g = gen ; anim = pronI.anim ;
      n = pronI.n ; nComp = pronI.nComp ; p = pronI.p ;
      pron = pronI.pron } ;

  mkNounPhrase : Number -> CommNounPhrase -> NounPhrase =
   \n,chelovek ->
    {s = \\cas => chelovek.s ! n ! (extCase cas) ;
     n = n ; g = chelovek.g ; p = P3 ; pron =False ;
     anim = chelovek.anim
    } ;
  pron2NounPhrase : Pronoun -> Animacy -> NounPhrase =
  \ona, anim ->
    {s = ona.s ; n = ona.n ; g = pgen2gen ona.g ;
     pron = ona.pron; p = ona.p ; anim = anim } ;
```

## D.3 Determiners

Determiners (only determinative pronouns in Russian) are inflected according to the gender of nouns they determine. The determined noun has the case parameter specific for the determiner:

```
Determiner : Type = Adjective ** { n: Number; c : Case } ;
```

```
anyPlDet = kakojNibudDet ** {n = Pl;  c= Nom} ;

detNounPhrase : Determiner -> CommNounPhrase -> NounPhrase =
\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 = okhotnik.g ;
   anim = okhotnik.anim
  } ;

indefNounPhrase : Number -> CommNounPhrase -> NounPhrase =
\n,mashina ->
  {s = \\c =>  mashina.s ! n !  (extCase c) ;
   n = n ; p = P3 ; g = mashina.g ; anim = mashina.anim ;
   pron = False
  } ;

defNounPhrase : Number -> CommNounPhrase -> NounPhrase =
\n,mashina ->
  { s = \\c => mashina.s ! n ! (extCase c) ;
   n = n ; p = P3 ; g = mashina.g ;anim = mashina.anim ;
   pron = False } ;
```

Genitives of noun phrases can be used like determiners, to build noun phrases. The number argument makes the difference between *мой дом - мои дома* (*my house - my houses*).

The variation like in *the car of John / John's car* in English is not equally natural for proper names and pronouns and the rest of nouns. Compare *дверца машины* and *машины дверца*, while *Ванина мама* and *мама Бани* or *моя мама* and *mama тоуа*. Here we have to make a choice of a universal form, which will be *моя мама - Вани мама - машины дверца* , which sounds the best for pronouns, a little worse for proper names and the worst for the rest of

nouns. The reason is the fact that possession/genetive is more a human category and pronouns are used very often, so we try to suit this case in the first place.

```
npGenDet : Number -> NounPhrase -> CommNounPhrase -> NounPhrase =
   \n,masha,mashina ->
        {s = \\c => case masha.pron of
             { True =>  masha.s ! (mkPronForm Nom No
               (Poss (gNum mashina.g n))) ++
                     mashina.s ! n ! (extCase c)  ;
               False => masha.s ! (mkPronForm Gen No
               (Poss (gNum mashina.g n))) ++
                     mashina.s ! n ! (extCase c)
             } ;
         n = n ; p = P3 ;  g = mashina.g ; anim = mashina.anim ;
         pron = False
        } ;
```

# D.4 Adjectives

## D.4.1 Simple adjectives

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

```
  oper

    extAdjective : AdjDegr -> Adjective = \adj ->
      { s = \\af => adj.s ! Pos ! af } ;

    extAdjFromSubst: CommNoun -> Adjective = \ vse ->
      {s = \\af => vse.s ! SF (numAF af) (caseAF af) } ;
```

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

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

### D.4.2 Adjective phrases

An adjective phrase may contain a complement, e.g. *моложе Риты*. Then it is used as postfix in modification, e.g. *человек, моложе Риты*.

```
IsPostfixAdj = Bool ;

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

Simple adjectives are not postfix:

```
adj2adjPhrase : Adjective -> AdjPhrase = \novuj ->
                            novuj ** {p = False} ;
```

### D.4.3 Comparison adjectives

Each of the comparison forms has a characteristic use:

Positive forms are used alone, as adjectival phrases (*высокий*).

```
positAdjPhrase : AdjDegr -> AdjPhrase = \bolshoj ->
  adj2adjPhrase (extAdjective bolshoj) ;
```

Comparative forms are used with an object of comparison, as adjectival phrases (*выше тебя*).

```
comparAdjPhrase : AdjDegr -> NounPhrase -> AdjPhrase =
\bolshoj, tu ->
  {s = \\af => bolshoj.s ! Comp ! af ++
       tu.s ! (mkPronForm Gen Yes NonPoss) ;
   p = True
  } ;
```

Superlative forms are used with a modified noun, picking out the maximal representative of a domain (*самый высокий дом*).

```
superlNounPhrase : AdjDegr -> CommNounPhrase -> NounPhrase =
 \bolshoj, dom ->
  {s = \\pf => bolshoj.s ! Super ! AF (extCase pf) dom.anim
       (gNum dom.g Sg) ++ dom.s  ! Sg ! (extCase pf) ;
   n = Sg ;
   p = P3 ;
   pron = False;
   anim = dom.anim ;
   g = dom.g
  } ;
```

### D.4.4 Two-place adjectives

A two-place adjective is an adjective with a preposition used before the complement. (Rem. `Complement = s2 : Preposition ; c : Case` ).

```
AdjCompl = Adjective ** Complement ;

complAdj : AdjCompl -> NounPhrase -> AdjPhrase =
\vlublen,tu ->
  {s = \\af => vlublen.s ! af ++ vlublen.s2 ++
        tu.s ! (mkPronForm vlublen.c No NonPoss) ;
   p = True
  } ;
```

### D.4.5 Complements

```
Complement =  {s2 : Preposition ; c : Case} ;

complement : Str -> Complement = \cherez ->
  {s2 = cherez ; c = Nom} ;

complementDir : Complement = complement [] ;

complementCas : Case -> Complement = \c ->
  {s2 = [] ; c = c} ;
```

## D.5 Individual-valued functions

An individual-valued function is a common noun together with the preposition prefixed to its argument (*ключ от дома*). The situation is analogous to two-place adjectives and transitive verbs.

We allow the genitive construction to be used as a variant of all function applications. It would definitely be too restrictive only to allow it when the required case is genitive. We don't know if there are counterexamples to the liberal choice we've made.

```
Function = CommNounPhrase ** Complement ;
```

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

```
appFunComm : Function -> NounPhrase -> CommNounPhrase =
\mama,ivan ->
   {s = \\n, cas =>
       mama.s ! n ! cas ++ mama.s2 ++
       ivan.s ! (mkPronForm mama.c No (Poss (gNum mama.g n)));
     g = mama.g ;
     anim = mama.anim
    } ;
```

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

```
funAsCommNounPhrase : Function -> CommNounPhrase = \x -> x ;

mkFun : CommNoun -> Preposition -> Case -> Function =
\f,p,c ->
  (n2n f) ** {s2 = p ; c = c} ;
```

The following is an aggregate corresponding to the original function application producing *децтво Ивана* and *Иваново децтво*. It does not appear in the resource abstract syntax any longer. Both versions return *децтво Ивана* although *Иваново децтво* must also be included. Such possesive form is only possible with proper names in Russian :

```
appFun : Bool -> Function -> NounPhrase -> NounPhrase =
\coll,detstvo, ivan ->
  let {n = ivan.n ; nf = if_then_else Number coll Sg n} in
  variants {
    defNounPhrase nf (appFunComm detstvo ivan) ;
                                  -- detstvoIvana
    npGenDet nf ivan detstvo
    } ;
```

The commonest cases are functions with Genitive.

```
funGen : CommNoun -> Function = \urovenCen ->
  mkFun urovenCen [] Gen ;
```

### D.5.1 Modification of common nouns

The two main functions of adjective are in predication (*Иван - молод*) and in modification (*молодой человек*). Predication will be defined later, in the chapter on verbs.

```
modCommNounPhrase : AdjPhrase -> CommNounPhrase ->
 CommNounPhrase = \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
  } ;
```

# D.6 Verbs

## D.6.1 Transitive verbs

Transitive verbs are verbs with a preposition for the complement, in analogy with two-place adjectives and functions. One might prefer to use the term *2-place verb*, since *transitive* traditionally means that the inherent preposition is empty and the case is accusative. Such a verb is one with a **direct object**. Note: Direct verb phrases where the Genitive case is also possible (*купить хлеба*, *не читать газет*) are overlooked in mkDirectVerb and can be expressed via more a general rule mkTransVerb.

```
TransVerb : Type = Verb ** {s2 : Preposition ; c: Case } ;

complementOfTransVerb : TransVerb -> Complement =
 \v -> {s2 = v.s2 ; c = v.c} ;
verbOfTransVerb       : TransVerb -> Verb   = \v ->
  {s = v.s; t = v.t; a = v.a ; v = v.v } ;

mkTransVerb : Verb -> Preposition  -> Case -> TransVerb =
 \v,p,cas -> v ** {s2 = p ; c = cas } ;

mkDirectVerb : Verb -> TransVerb = \v ->
  mkTransVerb v nullPrep Acc;

nullPrep : Preposition = [] ;
```

The rule for using transitive verbs is the complementization rule:

```
complTransVerb : Bool -> TransVerb -> NounPhrase ->
  VerbPhrase = \b,se,tu ->
   {s = se.s ; a = se.a ; t = se.t ; v = se.v ;
    s2 = negation b ;
    s3 = \\_,_ => se.s2 ++
        tu.s ! (mkPronForm se.c No NonPoss) ;
```

```
    negBefore = True } ;
```

### D.6.2 Verb phrases

Verb phrases are discontinuous: the parts of a verb phrase are (s) an inflected verb, (s2) verb adverbials (such as negation), 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} ;
```

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.

```
predVerb : Bool -> Verb -> VerbPhrase = \b,vidit ->
  vidit ** {
   s2 = negation b ;
   s3 = \\_,_ => [] ;
   negBefore = True
   } ;

negation :  Bool -> Str = backslashb ->
 if_then_else Str b [] "не";
```

Sometimes we want to extract the verb part of a verb phrase.

```
verbOfPhrase : VerbPhrase -> Verb = \v ->
  {s = v.s; t = v.t ; a = v.a ; v =v.v} ;
```

Verb phrases can also be formed from adjectives (- *молод*), common nouns (- *человек*), and noun phrases (- *самый молодой*). The third rule is overgenerating: - *каждый человек* has to be ruled out on semantic grounds. Note: in some case we can even omit a dash - : *я думаю, что он хороший человек.*

```
predAdjective : Bool -> Adjective -> VerbPhrase = \b,zloj ->
  { s= \\_ => "-" ;
    t = Present ;
    a = Imperfective ;
    v = Act ;
    s2 = negation b ;
    s3 = \\g,n => case n of {
        Sg => zloj.s ! AF Nom Animate (ASg g) ;
        Pl => zloj.s ! AF Nom Animate APl
        } ;
   negBefore = False
```

```
  } ;

predCommNoun : Bool -> CommNounPhrase -> VerbPhrase =
\b,chelovek ->
 {  s= \\_ => "-" ;
    t = Present ;
    a = Imperfective ;
    v = Act ;
    s2 = negation b ;
    s3 = \\_,n => (indefNounPhrase n chelovek ).s !
                  (mkPronForm Nom No NonPoss) ;
    negBefore = False
   } ;

predNounPhrase : Bool -> NounPhrase -> VerbPhrase =
 \b,masha ->
  { s= \\_ => "-" ;
    t = Present ;
    a = Imperfective ;
    v = Act ;
    s2 = negation b ;
    s3 = \\_,_ => masha.s ! (mkPronForm Nom No NonPoss) ;
    negBefore = False
  } ;

-- A function specific for Russian :
predNeedShortAdjective: Bool -> NounPhrase ->
CommNounPhrase -> Sentence = \ b, Jag, Dig -> { s =
  let {
    mne  = Jag.s ! (mkPronForm Dat No NonPoss) ;
    nuzhen  = need.s ! AF Nom Inanimate (gNum Dig.g Sg)  ;
    doctor = Dig.s ! Sg ! Nom ;
    ne = negation b
  } in
     mne ++ ne ++ nuzhen ++ doctor
  } ;
```

## D.7 Adverbials

```
adVerbPhrase : VerbPhrase -> Adverb -> VerbPhrase =
\poet, khorosho ->
  {s = \\vf => khorosho.s ++ poet.s ! vf ;
```

```
      s2 = poet.s2; s3 = poet.s3;
      a = poet.a; v = poet.v; t = poet.t ;
      negBefore = poet.negBefore } ;
```

Adverbials are typically generated by prefixing prepositions. The rule for creating locative noun phrases by the preposition *в* is a little shaky: *в России*, but *на острове*.

```
    locativeNounPhrase :  NounPhrase -> Adverb =
ivan ->
   {s = "в"Ъ иван.с ! (мкПронФорм Препос ЫЕс НонПосс)
    } ;
```

This is a source of the *man with a telescope* ambiguity, and may produce strange things, like *машины всегда*. Semantics will have to make finer distinctions among adverbials.

```
    advCommNounPhrase : CommNounPhrase -> Adverb ->
    CommNounPhrase = \chelovek,uTelevizora ->
      {s = \\n,c => chelovek.s ! n ! c ++ uTelevizora.s ;
       g = chelovek.g ;
       anim = chelovek.anim
      } ;
```

## D.8 Sentences

We do not introduce the word order parameter for sentences in Russian although there exist several word orders, but they are too specific to capture on the level we work here.

```
  oper
     Sentence : Type = { s : Str } ;
```

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

```
      predVerbPhrase : NounPhrase -> VerbPhrase -> Sentence =
      \Ya, tebyaNeVizhu -> { s =
         let
         { ya = Ya.s ! (mkPronForm Nom No NonPoss);
           ne = tebyaNeVizhu.s2;
           vizhu = tebyaNeVizhu.s ! VFin (gNum Ya.g Ya.n) Ya.p;
           tebya = tebyaNeVizhu.s3 ! Ya.g ! Ya.n
         }
         in
         if_then_else Str tebyaNeVizhu.negBefore
          (ya ++ ne ++ vizhu ++ tebya)
```

```
      (ya ++ vizhu ++ ne ++ tebya)

  } ;

-- A function specific for Russian:
U_predTransVerb : Bool -> TransVerb -> NounPhrase ->
NounPhrase -> Sentence = \b,Ser,Jag,Dig -> { s =
  let {
    menya  =  Jag.s ! (mkPronForm Gen Yes NonPoss) ;
    bolit  = Ser.s ! VFin (gNum Dig.g Dig.n) Dig.p ;
    golova = Dig.s ! (mkPronForm Nom No NonPoss) ;
    ne = negation b
  } in

    "y"++ menya ++ ne ++ bolit ++ golova } ;
```

This is a macro for simultaneous predication and complementation.

```
predTransVerb : Bool -> TransVerb -> NounPhrase ->
NounPhrase -> Sentence = \b,vizhu,ya,tu ->
  predVerbPhrase ya (complTransVerb b vizhu tu) ;
```

### D.8.1 Sentence-complement verbs

Sentence-complement verbs take sentences as complements.

```
SentenceVerb : Type = Verb ;
```

To generate *сказал, что Иван гуляет* / *не сказал, что Иван гуляет*:

```
complSentVerb :  Bool -> SentenceVerb -> Sentence ->
VerbPhrase = \b,vidit,tuUlubaeshsya ->
{s = vidit.s ; s2 = negation b ;
s3 = \_,_ => [" , что "] ++ tuUlubaeshsya.s ;
t = vidit.t ; v = vidit.v ; a = vidit.a ; negBefore = True } ;
```

## D.9 Sentences missing noun phrases

This is one instance of Gazdar's **slash categories**, 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**.

```
SentenceSlashNounPhrase = Sentence ** Complement ;

slashTransVerb : Bool -> NounPhrase -> TransVerb ->
  SentenceSlashNounPhrase = \b,ivan,lubit ->
     predVerbPhrase ivan (predVerb b (verbOfTransVerb lubit)) **
     complementOfTransVerb lubit ;
```

## D.10 Coordination

Coordination is to some extent orthogonal to the rest of syntax, and has been treated in a generic way in the module `CO` in the file `coordination.gf`. The overall structure is independent of category, but there can be differences in parameter dependencies.

### D.10.1 Conjunctions

Coordinated phrases are built by using conjunctions, which are either simple (*и*, *или*) or distributed (*как* - *так*, *либо* - *либо*).

The conjunction has an inherent number, which is used when conjoining noun phrases: *Иван и Маша поют* vs. *Иван или Маша поет*; in the case of *или*, the result is however plural if any of the disjuncts is.

```
Conjunction = CO.Conjunction ** {n : Number} ;
ConjunctionDistr = CO.ConjunctionDistr ** {n : Number} ;
```

## D.11 Relative pronouns and relative clauses

```
oper
  identRelPron : RelPron = { s  = \\gn, c, anim =>
     kotorujDet.s ! (AF c anim gn )} ;

  funRelPron : Function -> RelPron -> RelPron =
  \mama, kotoruj ->
    {s = \\gn,c, anim => let {nu = numGNum gn} in
           mama.s ! nu ! c ++
           mama.s2 ++ kotoruj.s !  gn ! mama.c ! anim
    } ;
```

Relative clauses can be formed from both verb phrases (*видит Машу*) and slash expressions (*я вижу*).

```
RelClause : Type = RelPron ;
```

```
relVerbPhrase : RelPron -> VerbPhrase -> RelClause =
 \kotoruj, gulyaet ->
  { s = \\gn, c, anim =>  let { nu = numGNum gn } in
    kotoruj.s ! gn ! c ! anim ++ gulyaet.s2 ++
    gulyaet.s ! VFin gn P3 ++ gulyaet.s3 ! genGNum gn ! nu
  } ;

relSlash : RelPron -> SentenceSlashNounPhrase -> RelClause =
 \kotoruj, yaVizhu ->
  {s = \\gn, _ , anim => yaVizhu.s2 ++
     kotoruj.s ! gn ! yaVizhu.c ! anim ++ yaVizhu.s
  } ;
```

A 'degenerate' relative clause is the one often used in mathematics, e.g. *число x, такое что x - чётное.*

```
relSuch :  Sentence -> RelClause =
A ->
  {s =
gn,c, anim => takoj.s !  AF c anim gn ++ "что"ЂЂ A.c } ;
```

The main use of relative clauses is to modify common nouns. The result is a common noun, out of which noun phrases can be formed by determiners. A comma is used before the relative clause.

```
modRelClause : CommNounPhrase -> RelClause -> CommNounPhrase =
 \chelovek,kotorujSmeetsya ->
  { s = \\n,c => chelovek.s ! n ! c ++ "," ++
    kotorujSmeetsya.s ! gNum chelovek.g n ! Nom ! chelovek.anim;
    g = chelovek.g ;
    anim = chelovek.anim
  } ;
```

## D.12 Interrogative pronouns

If relative pronouns are adjective-like, interrogative pronouns are noun-phrase-like. Actually we can use the very same type!

```
IntPron : Type = NounPhrase ;
```

In analogy with relative pronouns, we have a rule for applying a function to a relative pronoun to create a new one. We can reuse the rule applying functions to noun phrases!

```
funIntPron : Function -> IntPron -> IntPron =
  appFun False ;
```

There is a variety of simple interrogative pronouns: *какая машина*, *кто*, *что*.

```
nounIntPron : Number -> CommNounPhrase -> IntPron = \n, x ->
  detNounPhrase (kakojDet ** {n = n;  c= Nom}) x ;

intPronKto : Number -> IntPron = \num ->
{ s = table {

PF Nom _ _ => "кто";
PF Gen _ _ => "кого";
PF Dat _ _ => "кому";
PF Acc _ _ => "кого";
PF Inst _ _ => "кем";
PF Prepos _ _ => "ком"


  } ;
  g = Masc ;
  anim = Animate ;
  n = num ;
  p = P3  ;
  pron = False
} ;

intPronChto : Number -> IntPron = \num ->
{ s = table {


PF Nom _ _ => "что";
PF Gen _ _ => "чего";
PF Dat _ _ => "чему";
PF Acc _ _ => "что";
PF Inst _ _ => "чем";
PF Prepos _ _=> "чём"


  } ;
  g = Neut ;
  anim = Inanimate ;
  n = num ;
  p = P3  ;
  pron = False
} ;
```

## D.13 Utterances

By utterances we mean whole phrases, such as 'can be used as moves in a language game': indicatives, questions, imperative, and one-word utterances. The rules are far from complete.

N.B. we have not included rules for texts, which we find we cannot say much about on this level. In semantically rich GF grammars, texts, dialogues, etc, will of course play an important role as categories not reducible to utterances. An example is proof texts, whose semantics show a dependence between premises and conclusions. Another example is intersentential anaphora.

```
Utterance = SS ;

indicUtt : Sentence -> Utterance = \x ->
   postfixSS "." (defaultSentence x) ;
interrogUtt : Question -> Utterance = \x ->
   postfixSS "?" (defaultQuestion x) ;
```

## D.14 Questions

Questions are either direct (*ты счастлив?*) or indirect (*он спросил счастлив ли ты*).

```
param
  QuestForm = DirQ | IndirQ ;

oper
  Question = SS1 QuestForm ;
```

### D.14.1 Yes-no questions

Yes-no questions are used both independently (*ты взял мяч?*) and after interrogative adverbials (*почему ты взял мяч?*). Note: The particle *ли* can also be used in direct questions: видел ли ты что-нибудь подобное? but we are not considering this case.

```
 questVerbPhrase :  NounPhrase -> VerbPhrase -> Question =
  \tu,spish ->
  let { vu = tu.s !  (mkPronForm Nom No NonPoss);
  spish = spish.s !  VFin (gNum tu.g tu.n) tu.p
  ++ spish.s2 ++ spish.s3 !  tu.g !  tu.n } in
  { s = table {
  DirQ => vu ++ spish ;
  IndirQ => spish ++ "ли"++ vu
  } } ;
```

### D.14.2 Wh-questions

Wh-questions are of two kinds: ones that are like `NP - VP` sentences, others that are like `S/NP - NP` sentences.

```
intVerbPhrase : IntPron -> VerbPhrase -> Question =
\kto,spit ->
  {s = table { _  => (predVerbPhrase kto spit).s }
  } ;

intSlash : IntPron -> SentenceSlashNounPhrase -> Question =
\Kto, yaGovoru ->
  let {  kom = Kto.s ! (mkPronForm yaGovoru.c No NonPoss) ;
   o = yaGovoru.s2 } in
  {s = table {  _ => o ++ kom ++ yaGovoru.s  }
  } ;
```

### D.14.3 Interrogative adverbials

These adverbials will be defined in the lexicon: they include *когда* , *где*, *как*, *почему*, etc, which are all invariant one-word expressions. In addition, they can be formed by adding prepositions to interrogative pronouns, in the same way as adverbials are formed from noun phrases. N.B. we rely on record subtyping when ignoring the position component.

```
IntAdverb = SS ;
```

A question adverbial can be applied to anything, and whether this makes sense is a semantic question.

```
questAdverbial : IntAdverb -> NounPhrase -> VerbPhrase ->
Question = \kak, tu, pozhivaesh ->
  {s = \\q => kak.s ++ tu.s ! (mkPronForm Nom No NonPoss) ++
   pozhivaesh.s2 ++
   pozhivaesh.s ! VFin (gNum tu.g tu.n) tu.p ++
   pozhivaesh.s3 ! tu.g ! tu.n } ;
```

## D.15 Imperatives

We only consider second-person imperatives.

```
Imperative: Type = { s: Gender => Number => Str } ;

imperVerbPhrase : VerbPhrase -> Imperative = \budGotov ->
```

```
    {s = \\g, n => budGotov.s ! VImper n P2 ++ budGotov.s2 ++
     budGotov.s3 ! g ! n} ;

  imperUtterance : Gender -> Number -> Imperative -> Utterance =
   \g,n,I ->  ss (I.s ! g ! n ++ "!") ;
```

### D.15.1 Coordinating sentences

We need a category of lists of sentences. It is a discontinuous category, the parts corresponding to 'init' and 'last' segments (rather than 'head' and 'tail', because we have to keep track of the slot between the last two elements of the list). A list has at least two elements.

```
  ListSentence : Type = SD2 ;

  twoSentence : (_,_ : Sentence) -> ListSentence = CO.twoSS ;

  consSentence : ListSentence -> Sentence -> ListSentence =
    CO.consSS CO.comma ;
```

To coordinate a list of sentences by a simple conjunction, we place it between the last two elements; commas are put in the other slots, e.g. *ты куриш, вы пбете и я ем*.

```
  conjunctSentence : Conjunction -> ListSentence -> Sentence =
   \c,xs ->  ss (CO.conjunctX c xs) ;
```

To coordinate a list of sentences by a distributed conjunction, we place the first part (e.g. *как*) in front of the first element, the second part (*так и*) between the last two elements, and commas in the other slots. For sentences this is really not used.

```
  conjunctDistrSentence : ConjunctionDistr -> ListSentence ->
   Sentence = \c,xs ->
    ss (CO.conjunctDistrX c xs) ;
```

### D.15.2 Coordinating adjective phrases

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

```
  ListAdjPhrase : Type =
    {s1,s2 : AdjForm => Str ; p : Bool} ;

  twoAdjPhrase : (_,_ : AdjPhrase) -> ListAdjPhrase =
```

```
\x,y -> CO.twoTable AdjForm x y ** {p = andB x.p y.p} ;

consAdjPhrase : ListAdjPhrase -> AdjPhrase -> ListAdjPhrase =
  \xs,x ->
  CO.consTable AdjForm CO.comma xs x ** {p = andB xs.p x.p} ;

conjunctAdjPhrase : Conjunction -> ListAdjPhrase -> AdjPhrase
 = \c,xs -> CO.conjunctTable AdjForm c xs ** {p = xs.p} ;

conjunctDistrAdjPhrase : ConjunctionDistr -> ListAdjPhrase ->
 AdjPhrase = \c,xs ->
  CO.conjunctDistrTable AdjForm c xs ** {p = xs.p} ;
```

### D.15.3 Coordinating noun phrases

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.

```
ListNounPhrase : Type = { s1,s2 : PronForm =>
  Str ; g: Gender ; anim : Animacy ; n : Number ;
  p : Person ;  pron : Bool } ;

twoNounPhrase : (_,_ : NounPhrase) -> ListNounPhrase =
 \x,y ->
  CO.twoTable PronForm x y ** {n = conjNumber x.n y.n ;
  g = conjGender x.g y.g ; p = conjPerson x.p y.p ;
  pron = conjPron x.pron y.pron ;
  anim = conjAnim x.anim y.anim } ;

consNounPhrase : ListNounPhrase -> NounPhrase ->
 ListNounPhrase = \xs,x ->
  CO.consTable PronForm CO.comma xs x **
  {n = conjNumber xs.n x.n ; g = conjGender x.g xs.g ;
   anim = conjAnim x.anim xs.anim ;
   p = conjPerson xs.p x.p; pron = conjPron xs.pron x.pron} ;

conjunctNounPhrase : Conjunction -> ListNounPhrase ->
 NounPhrase = \c,xs ->
  CO.conjunctTable PronForm c xs **
    {n = conjNumber c.n xs.n ; anim = xs.anim ;
     p = xs.p; g = xs.g ; pron = xs.pron} ;

conjunctDistrNounPhrase : ConjunctionDistr ->
```

```
ListNounPhrase -> NounPhrase = \c,xs ->
  CO.conjunctDistrTable PronForm c xs **
   {n = conjNumber c.n xs.n ; p = xs.p ;
    pron = xs.pron ; anim = xs.anim ; g = xs.g } ;
```

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

```
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*, but *либо я, либо ты пойдешь*). 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* (Genetive 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*, *Антон или Олег - красивый*, *Маша или Олег - красивый* . The later is not totally correct, but there is no correct way to say that.

```
conjGender : Gender -> Gender -> Gender = \_,m -> m ;

conjAnim : Animacy -> Animacy -> Animacy = \_,m -> m ;
```

## D.16 Subjunction

Subjunctions (*когда*, *если*, etc) are a different way to combine sentences than conjunctions. The main clause can be a sentence, an imperative, or a question, but the subjoined clause must be a sentence.

There are uniformly two variant word orders, e.g. *если ты закуришь, я рассержусь* and *я рассержусь, если ты закуришь*.

```
Subjunction = SS ;

subjunctSentence : Subjunction -> Sentence -> Sentence
-> Sentence = \if, A, B ->
  ss (subjunctVariants if A.s B.s) ;

subjunctImperative : Subjunction -> Sentence -> Imperative
 -> Imperative =
  \if, A, B ->
  {s = \\g,n => subjunctVariants if A.s (B.s ! g ! n)} ;

subjunctQuestion : Subjunction -> Sentence -> Question ->
Question = \if, A, B ->
  {s = \\q => subjunctVariants if A.s (B.s ! q)} ;

subjunctVariants : Subjunction -> Str -> Str -> Str =
\if,A,B ->
 variants {if.s ++ A ++ "," ++ B ; B ++ "," ++ if.s ++ A} ;
```

## D.17 One-word utterances

An utterance can consist of one phrase of almost any category, the limiting case being one-word utterances. These utterances are often (but not always) in what can be called the default form of a category, e.g. the nominative. This list is far from exhaustive.

```
useNounPhrase : NounPhrase -> Utterance = \masha ->
  postfixSS "." (defaultNounPhrase masha) ;

useCommonNounPhrase : Number -> CommNounPhrase ->
Utterance = \n,mashina ->
  useNounPhrase (indefNounPhrase n mashina) ;

useRegularName : Gender -> SS -> NounPhrase =
\g, masha ->
  nameNounPhrase (case g of
  { Masc => mkProperNameMasc masha.s Animate;
      _ => mkProperNameFem masha.s Animate }) ;
```

Here are some default forms.

```
defaultNounPhrase : NounPhrase -> SS = \masha ->
  ss (masha.s ! PF Nom No NonPoss) ;
```

```
defaultQuestion : Question -> SS = \ktoTu ->
  ss (ktoTu.s ! DirQ) ;

defaultSentence : Sentence -> Utterance = \x -> x ;
```

# Bibliography


[BDL00] C. Brun, M. Dymetman, and V. Lux. Document structure and multilingual authoring. In *INLG'2000, Mitzpe Ramon, Israël*, pages 24–31, 2000.

[Ber96] K. D. Berndt. Protein Secondary Structure, 1996.

[Bio03] Bioinformatics.Org. Software map, 2003. `bioinformatics.org`.

[CM95] G. Cousineau and M. Mauny. *Approche fonctionnelle de la programmation*. Ediscience(Collection Informatique), Paris, 1995.

[Coq96] T. Coquand. An algorithm for type checking dependent types. *Science of Computer Programming*, 26:167–177, 1996.

[DLR00] M. Dymetman, V. Lux, and A. Ranta. XML and multilingual document authoring: Convergent trends. In *COLING, Saarbrücken, Germany*, pages 243–249, 2000.

[GM96] G.Hutton and E. Meijer. Monadic parser combinators. Technical report, University of Nottingham, Department of Computer Science, 1996. `http://www.cs.nott.ac.uk/~gmh//monparsing.ps`

[Hal03] T. Hallgren. Home Page of the Proof Editor Alfa. `http://www.cs.chalmers.se/~hallgren/Alfa`, 2003.

[HJR02] 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.

[Hue97] G. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[J.K02] J.Khegai. Functional parsing for biosequence analysis. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2002.

[J.K03] J.Khegai. Java GUI syntax editor for GF. In *The Joint Winter Meeting of Computing Science and Computer Engineering*. Chalmers University of Technology, 2003.

[JM00] D. Jurafsky and J. Martin. *Speech and language processing*. Prentice Hall, 2000.

[J.W00] J.Waldispuhl. Étude des qualités combinatoire du repliement des proteines, Rapport de DEA. École Polytechnique, Laboratoire d'informatique, 2000.

[Khe02] J. Khegai. java GUI Syntax Editor manual. `www.cs.chalmers.se/~aarne/GF/doc/javaGUImanual/javaGUImanual.htm`, 2002.

[KJR03] 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.

[KNR03] 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.

[Lin95] J. Lindström. Summary on reduplication. LINGUIST List: Vol-6-52., 1995.

[Mag94] 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.

[Mar02] S. Marlow. Happy, The Parser Generator for Haskell, 2002. `haskell.cs.yale.edu/happy`.

[MDP+00] M.Rayner, D.Carter, P.Bouillon, V.Digalakis, and M.Wirén. *The spoken language translator*. Cambridge University Press, 2000.

[MN94] 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.

[PS98] R. Power and D. Scott. Multilingual authoring using feedback texts. In *COLING-ACL 98*, Montreal, Canada, 1998.

[PSE98] R. Power, D. Scott, and R. Evans. Generation as a solution to its own problem. In *INLG'98*, Niagara-on-the-Lake, Canada, 1998.

[PSH03] R. Power, D. Scott, and A. Hartley. Multilingual generation of controlled languages. In *EAMT/CLAW-03*, Dublin, Irland, 2003.

[Pul84] I.M. Pulkina. *A Short Russian Reference Grammar*. Russky Yazyk, Moscow, 1984.

[Ran02] A. Ranta. The GF Resource grammar library, 2002. `http://tournesol.cs.chalmers.se/aarne/GF/resource/`.

[Ran03] A. Ranta. GF Homepage, 2003. `www.cs.chalmers.se/~aarne/GF/`.

[Ranar] A. Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *The Journal of Functional Programming*, to appear.

[She00] M.A. Shelyakin. *Spravochnik po russkoj grammatike (in Russian)*. Russky Yazyk, Moscow, 2000.

[Tea02] Functional Morphology Development Team. Open-Source Functional Morphology, 2002. `www.cs.chalmers.se/~aarne/morphology/`.

[Tea03] Gramlets Development Team. Gramlets Homepage, 2003. `www.cs.chalmers.se/~krijo/gramlets.html`.

[vDP00] K. van Deemter and R. Power. Multimedia document authoring using wysiwym editing. In *INLG-2000*, pages 15–19, Mitzpe Ramon, Israël, 2000.

[Wad00] T. Wade. *A Comprehensive Russian Grammar*. Blackwell Publishing, 2000.

[Y. 99] Y. Bertot. The CtCoq System: Design and Architecure. *Formal Aspects of Computing*, 11:225–243, 1999.

[Zau03] Sharp Zaurus. Sharp Zaurus Homepage, 2003. `www.myzaurus.com`.