Contract No.	FP7-ICT247914
Project full title	MOLTO - Multilingual Online Translation
Deliverable	D2.3 Grammar Tool Manual and Best Practices
Distribution level	Public
Contractual date of delivery	30 June 2012
Actual date of delivery	30 June 2012
Type	Prototype
Status version	V1.0
Authors	Aarne Ranta, John Camilleri, Grégoire Détrez, Ramona Enache, Thomas Hallgren
Task responsible	UGOT
Other contributors	Olga Caprotti, Dana Dannélls, Inari Listenmaa, Jordi Saludes

This document is mainly about the best practices. It gives a summary of the tools, but the user manuals proper are available as separate publications:

- J. Camilleri. An IDE for the Grammatical Framework, *Proceedings of FreeRBMT12*, to appear, 2012.
  http://www.molto-project.eu/sites/default/files/freerbmt2012.pdf
  Paper on the GF-Eclipse plug-in.

- A. Ranta, T. Hallgren, et al. *Grammar IDE*, MOLTO Deliverable D2.2, September 2011.
  http://www.molto-project.eu/sites/default/files/http:www.molto-project.eu:print:book:export:html:1379.pdf
  Documentation of the cloud-based IDE.

There is also a MOLTO publication directly related to best practices; some parts of it are summarized in this document:

- G. Détrez, R. Enache, and A. Ranta. Controlled Language for Everyday Use: the MOLTO Phrasebook. Fuchs & Rosner (eds), *CNL 2010 proceedings*, Springer LNCS/LNAI vol. 7175.
  http://www.molto-project.eu/sites/default/files/everyday.pdf

Contents

1 Introduction

This is a guide for building multilingual grammar-based translation systems in GF, Grammatical Framework (http://grammaticalframework.org). It is based on the experience gained in the European MOLTO project (Multilingual On-Line Translation, http://www.molto-project.eu).

MOLTO's mission is to enable **producer-oriented** translation, which means that providers of information can rely on the translations so much that they can publish them. This is contrasted to **consumer-oriented** translation, which is applied by consumers to get a rough idea of what the original document is about. The consumer scenario is much more common in current machine translation, exemplified by tools such as Google translate and Microsoft Bing. The term **dissemination** is often used for producers' translation, whereas consumers' translation is called **assimilation**.

The defining difference between the producer and consumer scenarios is **responsibility**: in consumer-oriented translation, it lies on the consumer. For instance, if a consumer uses Google translate to render a French e-commerce web page to Swedish, neither the French e-commerce site nor Google is responsible for the translation. Thus if a price originally indicated *99 euros* is translated as *99 kronor*—a typical error resulting from word alignments—the consumer cannot claim to get the product for this price (which corresponds to 11 euros). However, if the French e-commerce site itself has published the price *99 kronor* on its Swedish web site, then the responsibility lies on the producer.

It is well known since a long time that producer quality is difficult, if not impossible, to achieve in automatic translation. Bar-Hillel (1964) showed by simple examples that **word-sense disambiguation** may require unlimited world knowledge and intelligence. His examples used the English word *pen*, which may mean either a writing utensil (e.g. a fountain pen) or an enclosure where children play. In most languages, different words are used for these two senses—for instance, *penna* vs. *lekhage* in Swedish. Now, different senses of *pen* are probably involved in the sentences *the pen is in the box* and *the box is in the pen*. But constructing a computer program that finds out the correct sense can be arbitrarily hard. Bar-Hillel's conclusion was that FAHQT, **Fully Automatic High-Quality Translation** is impossible, "not only in foreseeable future, but in principle".

Nevertheless, FAHQT is what MOLTO aims for. What makes this possible is the use of **restricted language**. While consumer tools, such as Google translate, have to cope with any document that is thrown at them, producer tools can assume a limited fragment of language. For instance, e-commerce sites can be restricted to translating product descriptions, order forms, and customer agreements. MOLTO's mission is to make this not only possible but feasible for different users and scenarios, and scalable from small fragments of few languages to larger fragments of large numbers of simultaneous languages.

The diagram in Figure 1 illustrates the goals of MOLTO compared with the goals of consumer-oriented tools such as Google translate. The upper right corner is the ideal of full-coverage FAHQT, which no current system is even close to achieving. Google translate is approaching the ideal by adding precision; full coverage is assumed from the beginning. MOLTO is approaching it by adding coverage; full precision is assumed from the beginning.

The logarithmic scale on the x-axis indicates the number of "concepts" that are covered. Concepts are the smallest units of translation, such as words, multi-word terms, templates, and constructions. We don't need a precise technical definition of a "concept" to give an idea of the orders of magnitude. Thus MOLTO is scaling up its translation technology from hundreds to thousands of concepts, but it will still be two or three orders of magnitude behind full-coverage consumer tools.

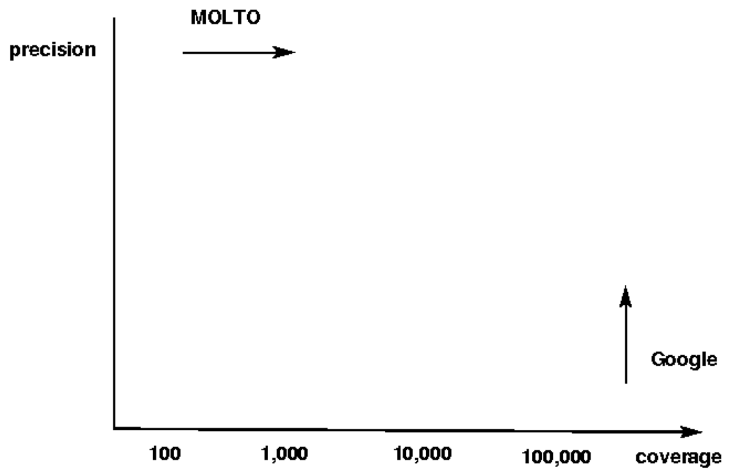


Figure 1: Orthogonal ways to approximate full precision and full coverage.

1.1 What this document is not about

This document is not an introduction to GF, but assumes basic knowledge of the GF grammar formalism, which can be gained for instance from the GF book (Ranta 2011) or the on-line tutorials at the GF web page, http://grammaticalframework.org. We assume skills in GF programming on the level of the GF book (Ranta 2011), Chapters 2 to 5. This is also the level assumed for GF application grammarians in general—that is, programmers assigned to build MOLTO-stype translation systems. Section 3 gives more details about the skills.

In addition to pure grammar-based systems, MOLTO project also explores **hybrid systems**, where grammars are used for improving the quality of statistical systems. Such hybrid systems use MOLTO tools in the consumers' (assimilation) scenario. They are, however, outside the scope of this documents, partly because they are so experimental that no best practices have developed yet. They are documented under MOLTO's WP5 and WP7.

The **evaluation** of the tools and best practices, based on experiences from MOLTO's case studies, is carried out in WP9 and will be presented in D9.2 (MOLTO Evaluation and Assessment Report).

2 A quick example

Before going to details, let us look at a small translation system built by MOLTO tools. Building the application consists of two steps:

1. Writing a multilingual grammar.

2. Setting up a web interface.

Only the first step requires manual programming work. The second step can be carried out by using one of MOLTO's standard interfaces. Of course, more work can be invested in modifying them to different purposes.

The quick example is a small fragment of the MOLTO phrasebook (Détrez et al., 2012). It enables the user to translate phrases like *the bar is open* between English, Finnish, and French. Figure 2 shows the entire code of the grammar, and Figure 3 a web application.

2.1 Writing a grammar

MOLTO translation is implemented in **GF**, the **Grammatical Framework** (Ranta 2011). A GF program is a **multilingual grammar**, which has two kinds of modules:

- an **abstract syntax** defining the **meanings** to be translated
- a set of **concrete syntaxes** mapping between meanings and languages

The abstract syntax works as an **interlingua** for translation. The mappings defined by concrete syntaxes are **reversible**: they can be used for both **parsing** (from language to meaning) and **linearization** (from meaning to language). A complete translation system for *n* languages needs *n*+1 modules: one abstract syntax, which is common, and *n* concrete syntaxes. Thus a system of the linear size $n + 1$ defines translations between the quadratic number $n(n - 1)/2$ of language pairs.

Figure 2.1 shows a grammar with

- two kinds of phrases (question, assertion)
- three kinds of places (bar, shop, station)
- two properties (open, closed)

It shows the abstract syntax together with English, Finnish, and French concrete syntaxes. The concrete syntaxes use the GF Resource Grammar Library. We will return to the details later.

2.2 Building a web application

Building a web application has three simple steps:

1. Compile your multilingual grammar to PGF. In the present case:

   ```
   $ gf -make ShopsEng.gf ShopsFin.gf ShopsFre.gf
   ```

2. Start the GF server. On a linux computer this can look as follows:

   ```
   $ gf -server
   ...
   Document root = /home/aarne/.cabal/share/gf-3.3.3/www
   Starting HTTP server, open http://localhost:41296/
   in your web browser.
   ```

3. Copy the PGF file from Step 1 to the `grammars` directory under the server's Document root:

   ```
   $ cp -p Shops.pgf /home/aarne/.cabal/share/gf-3.3.3/www/grammars/
   ```

After these steps are completed, you can use the translation applications on your server. For instance, opening "minibar" creates a "fridge magnet" view of translation by predictive parsing. For the Shops grammar, it may look as in Figure 2.3.

```
abstract Shops = {

flags startcat = Phrase ;

cat
  Phrase ;
  Statement ;
  Place ;
  Property ;

fun
  PQuestion  : Statement -> Phrase ;
  PAssertion : Statement -> Phrase ;
  SProperty  : Place -> Property -> Statement ;

  Bar, Shop, Station : Place ;
  Open, Closed : Property ;
}
```

```
--# -path=.:present

concrete ShopsEng of Shops = open SyntaxEng, ParadigmsEng in {

lincat
  Phrase = Text ;
  Statement = Cl ;
  Place = NP ;
  Property = AP ;

lin
  PQuestion  s = mkText (mkQS s) ;
  PAssertion s = mkText (mkS s) ;
  SProperty pl prop = mkCl pl prop ;

  Bar = mkPlace "bar" ;
  Shop = mkPlace "shop" ;
  Station = mkPlace "station" ;
  Open = mkProperty "open" ;
  Closed = mkProperty "closed" ;

oper
  mkPlace : Str -> NP = \s -> mkNP the_Det (mkN s) ;
  mkProperty : Str -> AP = \s -> mkAP (mkA s) ;
}
```

```
--# -path=.:present

concrete ShopsFin of Shops = open SyntaxFin, ParadigmsFin in {

lincat
  Phrase = Text ;
  Statement = Cl ;
  Place = NP ;
  Property = Adv ;

lin
  PQuestion  s = mkText (mkQS s) ;
  PAssertion s = mkText (mkS s) ;
  SProperty pl prop = mkCl pl prop ;

  Bar = mkPlace "baari" ;
  Shop = mkPlace "kauppa" ;
  Station = mkPlace "asema" ;
  Open = mkProperty "auki" ;
  Closed = mkProperty "kiinni" ;

oper
  mkPlace : Str -> NP = \s -> mkNP the_Det (mkN s) ;
  mkProperty : Str -> Adv = \s -> mkAdv s ;
}
```

```
--# -path=.:present

concrete ShopsFre of Shops = open SyntaxFre, ParadigmsFre in {

flags coding = utf8 ;

lincat
  Phrase = Text ;
  Statement = Cl ;
  Place = NP ;
  Property = AP ;

lin
  PQuestion  s = mkText (mkQS s) ;
  PAssertion s = mkText (mkS s) ;
  SProperty pl prop = mkCl pl prop ;

  Bar = mkPlace "bar" ;
  Shop = mkPlace "magasin" ;
  Station = mkPlace "gare" ;
  Open = mkProperty "ouvert" ;
  Closed = mkProperty "fermé" ;

oper
  mkPlace : Str -> NP = \s -> mkNP the_Det (mkN s) ;
  mkProperty : Str -> AP = \s -> mkAP (mkA s) ;
}
```

Figure 2: A multilingual grammar for shops.

2.3 Some key features

Linguistic knowledge. Even the most trivial natural language grammars involve expert linguistic knowledge. In the current example, we have, for instance, **word inflection** and **gender agreement** shown in French: *le bar est ouvert* ("the bar is open", masculine) vs. *la gare est ouverte* ("the station is open", feminine). As Step 3 in Figure 3 shows, the change of the noun (*bar* to *gare*) causes an automatic change of the definite article (*le* to *la*) and the adjective (*ouvert* to *ouvert*). Yet there is no place in the grammar code (Figure 2) that says anything about gender or agreement, and no occurrence of the words *la*, *le*, *ouverte*! The reason is that such linguistic details are inherited from a library, the GF **Resource Grammar Library** (RGL). The RGL guarantees that application programmers can write their grammars on a high level of abstraction, and with a confidence of getting the linguistic details automatically right.

Language differences. The RGL takes care of the rendering of linguistic structures in different languages. In the current example, these structures include building a clause from a place name and a property (`mkCl`) and building a place name from a noun with the definite article (`mkNP the_Det`). The renderings are different in different languages, so that e.g. the French definition of the constant `the_Det` produces a word whose form depends on the noun, whereas Finnish produces no article word at all. These variations, which are determined by the grammar of each language, are automatically created by the RGL. However, the example also shows another kind of variation: English and French use adjectives to express "open" and "closed", whereas Finnish uses adverbs. This variation is chosen by the grammarian, by picking different RGL types and categories for the same abstract syntax concepts. In GF, this variation is known as **compile-time transfer**. **Transfer**, in general machine translation theory, means the change of syntactic structure. Compile-time means that the transfer is performed when the grammar is compiled, which makes the resulting translation system more efficient at **run time**, i.e. when actual translations is performed.

3 When to use MOLTO tools

The first relevant question for the "best practices" is obviously:

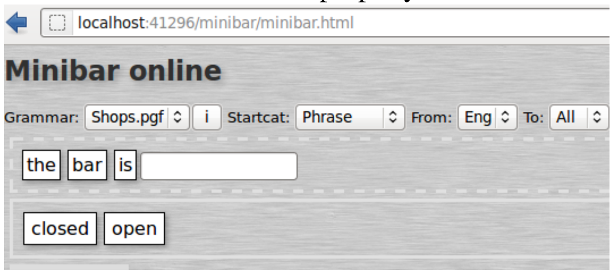
- *When should I use MOLTO technology rather than something else?*

The short answer is: when you want to translate with **full precision** and only need to deal with **maximally thousands of concepts**. Another indication is **multiple languages**: MOLTO's advantages are at their clearest when the system is scaled up from two languages to more. There is some initial overhead when just two languages are involved, but this gets amortized when more languages are added. Also the presence of **morphologically complex languages** is an indication for using the MOLTO approach, in particular if the languages are included in the resource grammar library (see next question).

- *What languages can be dealt with?*

GF has been put to work for at least a hundred different languages. But there are 25 languages that are distinctively the most well-prepared ones for applications. They are the ones included in the **RGL**, the GF **Resource Grammar Library**. Table 3 shows a summary of the RGL languages, together with what MOLTO applications have been built for each of them. In the table, the ISO 3-letter code is used in the RGL to indicate the file for each language (mostly ISO 639-2 B, with some exceptions). Smart and Dict stand for the lexicon resources available; the existence of either of them makes it easy to add lexical items to grammars. The rest of the columns are MOLTO applications built for that language. The more

1. Select a property.



localhost:41296/minibar/minibar.html

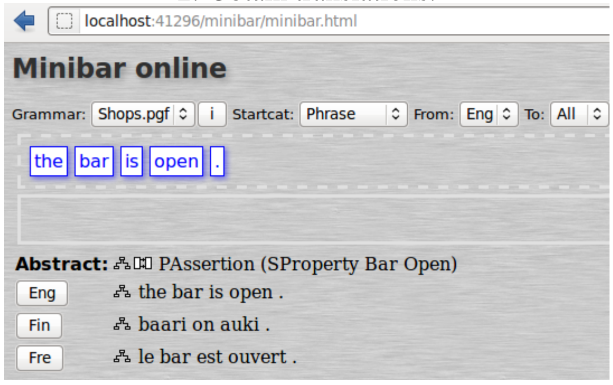
Minibar online

Grammar: Shops.pgf i Startcat: Phrase From: Eng To: All

the bar is

closed open

2. Obtain translations.



localhost:41296/minibar/minibar.html

Minibar online

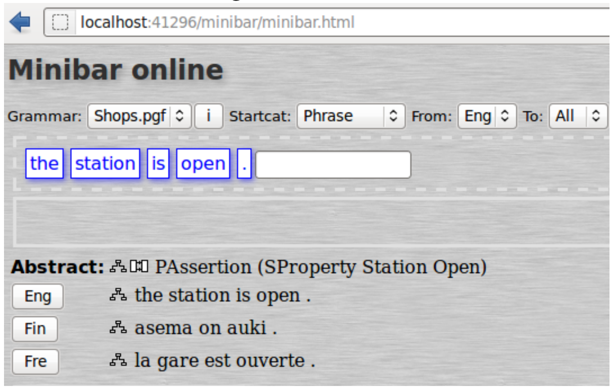
Grammar: Shops.pgf i Startcat: Phrase From: Eng To: All

the bar is open .

Abstract: PAssertion (SProperty Bar Open)

Eng	the bar is open .
Fin	baari on auki .
Fre	le bar est ouvert .

3. Change “bar” to “station”.



localhost:41296/minibar/minibar.html

Minibar online

Grammar: Shops.pgf i Startcat: Phrase From: Eng To: All

the station is open .

Abstract: PAssertion (SProperty Station Open)

Eng	the station is open .
Fin	asema on auki .
Fre	la gare est ouverte .

Figure 3: A web application with the Shops grammars.

Language	**Code**	**Smart**	**Dict**	**MOLTO**	**Phraseb**	**Math**	**Museum**	**Attempto**	**Patent**
Afrikaans	Afr	+							
Bulgarian	Bul		+	+	+	+			
Catalan	Cat	+		+	+	+			
Danish	Dan	+		+	+			+	
Dutch	Dut	+		+	+			+	
English	Eng	+	+	+	+	+	+	+	+
Finnish	Fin	+	+	+	+	+	+	+	
French	Fre	+	+	+	+	+	+	+	+
German	Ger	+	+	+	+	+		+	+
Hindi	Hin	+	+		+				
Italian	Ita	+		+	+	+	+	+	
Japanese	Jpn	+							
Latvian	Lav	+							
Nepali	Nep	+							
Norwegian	Nor	+		+	+				
Persian	Pes				+				
Punjabi	Pnb								
Polish	Pol			+	+	+			
Romanian	Ron	+		+	+	+			
Russian	Rus	+	+	+	+	+			
Sindhi	Snd	+							
Spanish	Spa	+		+	+	+		+	
Swedish	Swe	+	+	+	+	+	+	+	
Thai	Tha	+			+				
Urdu	Urd	+			+	+			

{ total | 25 | 21 | 8 | 15 | 19 | 13 | 5 | 9 | 3 |

Table 1: Languages in the GF RGL and their use in MOLTO. Smart = smart paradigms available, Dict = large dictionary available. MOLTO = the languages in MOLTO work plan. Phraseb(ook), Math, Museum, Attempto, and Patent are MOLTO applications.

Language	**Fluency**	**GF skills**	**Inf. dev.**	**Inf. testing**	**Ext. tools**	**RGL edits**	**Effort**
Bulgarian	###	###	-	-	-	#	##
Catalan	###	###	-	-	-	#	#
Danish	-	###	+	+	+	##	##
Dutch	-	###	+	+	+	#	##
English	##	###	-	+	-	-	#
Finnish	###	###	-	-	-	#	##
French	##	###	-	+	-	#	#
German	#	###	+	+	+	##	###
Italian	###	#	-	-	-	##	##
Norwegian	#	###	+	+	+	#	##
Polish	###	###	+	+	+	#	##
Romanian	###	###	-	-	+	###	###
Spanish	##	#	-	-	-	-	##
Swedish	##	###	-	+	-	-	##

Table 2: Development effort for the MOLTO Phrasebook.

applications there are, the better the language has been tested; languages with no applications are recent, and they can still contain bugs that are best revealed by applications.

Even if your target language is not included in the RGL, it may soon be. The RGL is an open-source project, which receives a few new languages every year.

- *How much time does it take to build a translation system?*

The typical size of current systems are in the order of hundreds of concepts. Getting such a system work for two languages might be a three days' work for someone who is fluent in GF programming. Adding a new language is a day or two for someone who knows the language in question. Table 2 shows a summary of the efforts for one of MOLTO's applications, the Phrasebook (Détrez et al. 2012).

- *What skills and training are needed?*

Typical GF programmers are persons with a programming background and an interest for languages. Two years' computer science education is certainly a sufficient background, but also education in mathematics or linguistics has worked.

GF itself can be learnt in less than a week. By using the RGL, a GF programmer can port a grammar to a language that she doesn't know with the help of a native speaker informant.

Adding a language to the RGL also requires theoretical knowledge of the language involved, but this can be gained from good grammar books; no training in theoretical linguistics is required, although it of course helps. But the very point of the RGL is that the *users* of it will not need theoretical knowledge of languages. Thus there is a main division of GF programmers to **resource grammarians**, who contribute to the RGL, and **application grammarians**, who just use the RGL. This document is written for application grammarians.

- *How can I learn GF?*

Before and during the MOLTO project, training events and tutorials have been given, with length ranging from three hours (e.g. the CADE tutorial in 2011) to two weeks (the GF Summer Schools in 2009 and 2011). Many programmers have also learnt GF from the GF book (Ranta 2011) or the on-line tutorial,

http://www.grammaticalframework.org/doc/tutorial/gf-tutorial.html

The tutorial is roughly parallel to the book, chapters 1 to 8. The concepts needed for application grammar writing are covered in the book chapters 2 to 5, where chapters 3 and 4 can be read in a cursory fashion. Doing the exercises parallel to reading, and testing them on a computer, is an essential part of the learning process.

- *What domains can the translations address?*

There is a technical flavour to many GF applications, resulting from GF's roots in logic and theorem proving. Thus there are GF grammars for mathematical proofs and exercises, and for software specifications. But this was extended to spoken dialogue systems in the TALK project (Bringert et al. 2005), and MOLTO has extended the use of GF to many new domains: tourist phrasebooks, museum object descriptions, decision making systems, and pharmaceutical patents. The general answer is: whatever can be described by a formal grammar is a possible domain of MOLTO translation.

- *What applications and use cases are there?*

MOLTO's explicit focus is on **web-based translation systems**. But these systems can also be ported to **mobile applications** on the Android platform.

- *What alternative tools are there?*

The main choice is probably between consumer and producer tools: if the application is a contumer tool meant to deal with just any input, then **SMT** (**Statistical Machine Translation**, Koehn 2010) is perhaps the way to go, or a system like Apertium (Forcada et al., 2011). GF is approaching this domain as well, by the development of **hybrid systems** where grammars are combined with statistics (Enache et al. 2012, Ahlberg and Enache 2012, Angelov 2012).

But this document has its focus on producer tools with high precision, so we assume the application needs "deep grammars". Then the main alternatives are HPSG, LFG, and Regulus. All these systems have been used for grammar-based translation, and they have equivalents of resource grammar libraries. Our feeling is that GF is better suited for engineering-oriented people and requires less linguistic training; but this document is not a place to carry out a comparison.

4 The choice of tools

The GF tools covered in this document are available from http://grammaticalframework.org under open-source licenses. The main tools are:

1. The GF grammar compiler, i.e. the program invoked by the command `gf` in an OS shell. It can be used in two ways,
   - as a batch compiler for preparing end-user products
   - as an interactive shell for testing grammars during development
2. GF IDE's (Integrated Development Environments)

- GF-Eclipse plug-in for desktop use
- GFSE, a cloud-based editor for GF grammars

3. Grammar diagnostic tools
   - displaying grammar information
   - statistics about a grammar
   - ambiguity checking
   - unit and regression testing
4. The GF Resource Grammar Library: morphology, syntax, and lexicon for 25 languages. Tools supporting the use of the library include
   - the RGL API synopsis
   - the RGL source code browser
   - the RGL application expression editor
5. The GF run-time system, i.e. an interpreter for PGF binaries (Portable Grammar Format), which are produced by the grammar compiler
   - PGF interpreter in Haskell, integrated in the GF compiler shell
   - PGF interpreter in Java, useful for Java applications such as Android
   - PGF interpreter in C, useful for large-volume and large-coverage applications, and also for iPhone applications
6. GF web application interfaces
   - a small-scale interactive translator with "fridge magnets"
   - a large-scale translator with post-editing support
   - s translation quiz application
   - a JavaScript library usable for custom interfaces
7. GF mobile application libraries
   - and Android library based on Java runtime
   - an iPhone library based on C runtime (forthcoming)

In this document, we will focus on the tools 1 to 4. An essential feature of GF grammars is that they can be used on multiple run-time systems and user applications. Hence the application builders can choose their engineering tools independently of the targeted applications.

4.1 The GF grammar compiler

The grammar compiler is available as both source code and binaries. For most developers of application grammars, the binary distributions are the best choice. They don't require any external tools or libraries, and are readily available for the major operating systems: Linux, Mac OS, and Windows. The present document has been tested with GF version 3.3.3 from March 2012. But GF developers are committed to backward compatibility, and hence grammars that have worked in old versions should continue to work in newer ones. If this is not the case, bug reports are welcome to the bug tracking system

http://code.google.com/p/grammatical-framework/issues/list

The source code of the grammar compiler is written in Haskell. It can be obtained as a package accompanying each binary distribution, but also via a version control system, **Darcs**, which contains the current developer code. The Darcs distrubution is mirrored in **GitHub**, which is another way to get up-to-date sources, but is read-only.

Using the source distribution requires other Haskell tools, for instance, the GHC compiler. The binary distributions don't require any Haskell tools.

The shell and the batch compiler are one and the same program, just called with different options. The basic way to use the shell is the OS command

```
$ gf
```

which opens an interactive session. The basic way to use the batch compiler is

```
$ gf -make <GrammarFiles>
```

which converts a set of source files into a multilingual PGF file, usable in run-time applications. Both commands have many further options; a summary of them can be seen with

```
$ gf -help
```

4.2 Development environments

4.2.1 Text editor + GF shell

This is the traditional development environment for GF, with Emacs the most popular text editor. **GF modes** are available for Emacs, Geany, and Gedit; see

http://www.grammaticalframework.org/doc/gf-editor-modes.html

As most languages use **non-ASCII characters**, it is important to verify that the editor and the operating system shell support **Unicode**, in particular, the UTF-8 encoding. Some Windows shells, for instance, lack this support, which means that unicode characters cannot be seen properly.

4.2.2 The GF-Eclipse plug-in

Eclipse is a state of the art IDE (Integrated Development Environment) initially designed for Java, but now suporting many languages. The GF plug-in supports

- Syntax highlighting and error detection
- Code folding, quick block-commenting, automatic code formatting
- Definition outlining, jump to declaration, find usage
- Warnings for problems in module dependency hierarchy
- Launch configurations, i.e. compilation directly from IDE
- Use of GF Shell from within Eclipse
- Auto-completion for declared identifiers

- Background compilation (shallow) using project builder
- Support for Open Declaration (F3), including qualified names
- Code generation for new languages in application grammars
- Inline documentation for function calls, overloads
- Proper cross-reference handling with qualified names
- Test management and testing tool
- External library browser

See

http://www.grammaticalframework.org/eclipse/index.html

for detailed on-line documentation. The Eclipse plug-in has beed adopted by MOLTO's industrial partners Ontotext and Be Informed, which use Eclipse as their general software development environment.

4.2.3 The cloud-based IDE

This is a development environment that requires no installation of software, but works in standard web browsers. It has mainly been used as a quick introduction to GF, but it also has the advantage of being integrated with the wider concept of the **GF cloud**,

http://cloud.grammaticalframework.org/

where translation and grammar development tools can be combined. The main use case is **changing a running translation system**, for instance, adding new words or rules. This functionality is a part of MOLTO's cloud-based translation environments, but it is also being adapted to the **Multilingual Semantic Wiki** (WP11) by University of Zurich.

The main limitations of the cloud-based IDE are currently the following:

- simplified module system: only grammars with one abstract syntax and one concrete syntax module per language are supported, which excludes complex module hierarchies
- simplified input language: only grammars created within the IDE are supported, which excludes the import of grammars created in other ways (except the RGL modules, which are available as libraries)

These limitations imply that the IDE is not yet suitable for large-scale grammar development projects. But it will be scaled up during the MOLTO project, and the **GF Compile Server** at will continue to support its use on http://grammaticalframework.org

More about the cloud-based IDE can be found in the on-line documentation

http://cloud.grammaticalframework.org/gfse/about.html

4.3 Grammar diagnostic tools

The grammar diagnostic tools are based on functionalities in the GF compiler. They are all available in the shell, and the programmer can write scripts to automate grammar testing. Some of the diagnostics also have an easy access via the GF-Eclipse plug-in. But let us have an overview of the commands that have proven useful in practice, by looking at the shell commands. Most of these commands are recommended for frequent use when developing a grammar. Some of them can also be used for generating statistics and documentation.

4.3.1 Compilation diagnostics

These are available by the use of options and flags to the import command (`import` or `i` in the GF shell) or the batch compiler (`gf` in the OS shell). Thus

```
import -v FILE
```

gives detailed information on the compilation phases. For instance, if the compilation takes a long time, this flag makes it possible to see the role of the different phases and different functions in it. It also shows the PGF target code size of each linearization rule.

```
import -src FILE
```

forces compilation from source, which means the possibility for more information

```
import -retain FILE
```

forces retaining the source grammar modules in the shell, which makes some diagnostics available after the compilation. It also prevents building the PGF run-time code, which makes compilation faster.

The batch-mode compiler also has a set of of **phase dumps**. For instance,

```
gf --dump-tc FILES
```

dumps the output of the type checking phase. See

```
gf --help
```

for a full list of dump options.

4.3.2 Grammar display modes

These modes are divided into those that work on the run-time PGF grammar, and those that work on source modules (either `.gf` or `.gfo` files). First the most important PGF diagnostics:

```
print_grammar -words
```

shows the complete list of terminal tokens in the current PGF grammar. This is a quick way to see for instance character encoding errors, faulty morphology, or typos.

```
print_grammar -printer=bnf
```

shows the grammar as a BNF (context-free) approximation. This can be *very* large.

```
print_grammar -printer=fa | write_file -file=autom.dot
```

builds a finite automaton approximating the grammar. The format is graphviz code, which is in this example saved into a file named `autom.dot`, from which a graphical image can be generated. These automata are often very large and therefore only feasible for small grammars.

```
print_grammar -missing
```

shows which functions have not been defined in different concrete syntaxes.

```
abstract_info EXP
```

provides information about a function, an expression, or a category in abstract syntax—for instance, the type of an expression, as well as its probability (which is the product of the probabilities of its nodes).

```
help print_grammar
help abstract_info
```

list many more options, but the most frequently used ones are above.

The `print_grammar` listings are very long, and it often makes sense to save them in a file, as in the finite automaton example above. But often the main interest is their size. This can be computed from inside the GF shell by using a **system pipe** (`?`), which feeds a GF command to an OS command. Thus

```
print_grammar -printer=bnf | ? wc -l
```

gives the number of BNF rules in the context-free approximation, whereas

```
print_grammar -words | ? wc -w
```

shows how many distinct tokens the grammar has.

The `print_grammar` and `abstract_info` commands work on fully compiled grammars, that is, the PGF format. Such a grammar is present in the **run-time mode** of the GF shell, which results from any successful `import` command without the `-retain` option. If this option has been used, the GF shell is in the **source mode** and contains a bunch of source modules, which can be read from binary GF object files (`.gfo`) in addition to textual source files (`.gf`).

The source mode enables inspecting the module dependency graph, which is written in a file and can be further processed by the graphviz `dot` command:

```
> dependency_graph
> ! dot -Tpng _gfdepgraph.dot > deps.png
```

The graph shows all modules inherited and opened via module headers. If the RGL library is used, this graph gets huge. The flag `-only` restricts it to a subset of modules whose names match the pattern:

```
dg -only=Phrasebook*,Greetings*,Sentences*,Words*,Syntax*,Paradigms*
```

The resulting graph is shown in Figure 4.3.2.

Another command enabled by the source mode of the GF shell is `compute_concrete`. A typical use case is to introduce a morphological paradigms module and test different inflection patterns:

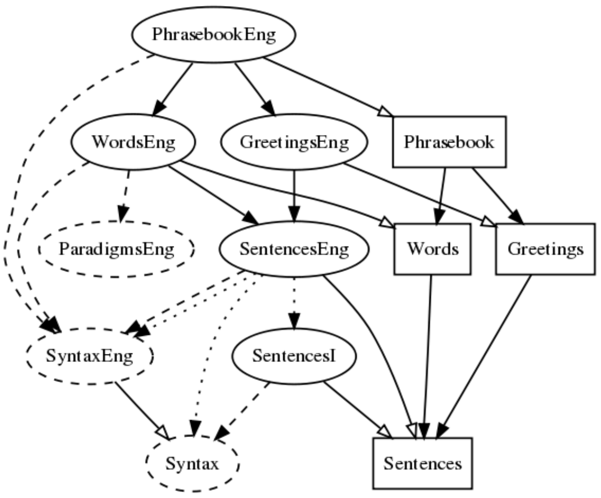


Figure 4: Module dependency graph for the MOLTO phrasebook.

```
> import -retain alltenses/ParadigmsGer.gfo

> compute_concrete -list mkN "Buch"
Buch, Buch, Buch, Buchs / Buches, Buche, Buche, Buchen, Buche,
ResGer.Masc

> compute_concrete -list mkN "Buch" "Bücher" neuter
Buch, Buch, Buch / Buche, Buchs / Buches, Bücher, Bücher, Büchern,
Bücher, ResGer.Neutr
```

The command `show_opers` shows the available library functions for a given type:

```
> show_opers N
mkN : Str -> N
mkN : Str -> Str -> Gender -> N
mkN : Str -> Str -> Str -> Str -> Str -> Str -> Gender -> N
```

4.3.3 Generation and testing

The grammar developer should maintain a number of test sets and scripts for continuous testing of the grammar. The most versatile format for this is a **treebank**—a set of abstract syntax trees, possibly with their linearizations. The GF-Eclipse plug-in has a method for creating and using treebanks. This method is ultimately based on the `-treebank` option of the linearization command in the GF shell. With this option, any tree can be turned into a treebank entry, that is, into a tree together with its linearizations.

Here is a simple example. Notice that it requires the use of GF in the run-time mode, because it performs linearization.

```
> i PhrasebookEng.gf PhrasebookGer.gf

> l -treebank (QProp (PropAction (ASpeak YouFamMale (LangNat German))))
```

```
Phrasebook: (QProp (PropAction (ASpeak YouFamMale (LangNat German))))
PhrasebookEng: do you speak German ?
PhrasebookGer: sprichst du Deutsch ?
```

The command thus expects a tree, which it converts to strings by linearization in every language that is in scope. The tree can be given explicitly, as here; this method is mainly useful when writing a treebank-producing script:

```
l -treebank (QProp (PropAction (ASpeak YouFamMale (LangNat German))))
l -treebank (QProp (PropAction (ASpeak YouPolMale (LangNat English))))
```

Such a script can be run over and over again, and the results can be compared for **regression testing**. The trees can be carefully chosen to represent all relevant constructs of the grammar, which amounts to **unit testing**.

Trees can also be created by **exhaustive generation**, which can be piped to treebanking,

```
generate_trees | l -treebank
```

The GF shell command `help generate_trees` lists the options and flags of the command; trees can be restricted by depth, to a certain number, to a certain category (other than the `startcat` of the grammar), and, as the most advanced option, with **tree patterns**, which are trees with **metavariables** (`?`):

```
generate_trees PQuestion (QProp (PropAction (ASpeak ? ?)))
```

An alternative is **random generation** (`generate_random`), which has all the options of exhaustive generation, but can also be biased with **constructor probabilities**:

```
generate_random -probs=FILE
```

See `help generate_random` for more options.

One common use of random generation is **ambiguity testing**:

```
generate_random | linearize -tr | parse
```

This pipe shows a random string covered by the grammar and all its parses. There is ongoing work on more tool support for ambiguity testing, expected to deliver by the end of 2012.

4.4 Resource grammar tools

4.4.1 The structure of the resource grammar

The Resource Grammar Library, RGL, has currently 902 modules, which means 36 modules per language. The sheer number of 902 would be totally unmanageable without a clear structure.

The most important structural feature is **language transparency**. This means that most modules *M* exist in 26 copies, where one has the name *M* (an abstract syntax or an interface) an the rest have the name *ML*, where *L* is the 3-letter language code (Table 1). Thus the module names are for the most part combined from about 36 topic names and 25 language names. Moreover, for any languages *L* and *K*, *ML* and *MK* have *M* as their common interface, which technically is either an `abstract` or an `interface`.

Another feature, almost equally important, is **layering**. The RGL has two layers: the API and the internals. The API for each language has just a few modules, which have uniform names across languages. The rest belongs to the internals and needs only be known by the RGL implementors.

Simple applications of the RGL only need to import two kinds of modules, both from the API:

- `Syntax`, syntactic categories and combination rules,
- `Paradigms`, morphological functions for lexicon building.

For example, the code in Figure 2 only imports these two.

Sometimes, however, a handful of other models are needed. Still a part of the API layer, the RGL has

- `Symbolic`, functions for mixing text with formulas,
- `Irreg`, a list of irregular words (mostly verbs) for some languages,
- `Dict`, a large-scale morphological dictionary,
- `Try`, a combination of `Syntax`, `Paradigms`, and `Lexicon`, useful for testing RGL function combinations in the GF shell, or in strictly monolingual code.

In addition to the API layer, some modules from the internal layer are sometimes needed in more advanced applications:

- `Extra`, language-specific extensions of the RGL,
- `Grammar`, direct access to the internal RGL syntax.

The only valid reason for using these modules is that sometimes a required structure is missing from the common API, i.e. `Syntax`. In the case of `Extra`, it is a structure that will never be available in the common API. In the case of `Grammar`, it is a structure intended to be common, but not yet implemented in all languages. Thus the latter case has to do with transient problems of the library. The former case is really a flaw in the current RGL: there *should* be an API layer access to the `Extra` modules. This flaw will be removed in later versions of the RGL.

A couple more modules are sometimes used in applications:

- `Res`, a low-level resource for the RGL internals, defininining the basic combinatorics and syntactic types,
- `Morpho`, another low-level RGL-internal resource, mostly defining paradigms for structural words.

Neither of these modules should ever be used in application grammars. They are changing constantly, and are likely to break any program that uses them. The reason why they are sometimes needed is that the RGL lacks some relevant structure. The recommendation is to report this to the RGL developers. If this is not quick enough, the application programmer should create her `ExtraPlus` module, which is compatible with the RGL so that it can be merged with the regular `Extra` module with that language, and which is the only module that makes reference to the low-level modules.

In summary, the following principle could not be stressed too much:

Importing modules below the API-layer implies a high risk of breaking the program in the future, because the RGL internals are not committed to backward-compatibility.

This is the most important lesson learned from an earlier RGL application, the WebALT project (2005–2006). In the beginning, the RGL version was 0.9. There was no separate API layer, and the internal modules were used directly. Before the release of RGL 1.0, the internals were changed quite a lot, which was in itself an improvement, but broke the WebALT code that had relied on RGL 0.9. The lesson learned by RGL developers was to create a separate API layer. The users had to learn to use this layer and not the internals.

4.4.2 The RGL code

The RGL modules are available both as source and binary. The standard GF binary distribution contains the binary (`.gfo`) files, which are accessible in GF relative to the path defined by the variable `GF_LIB_PATH`. They come in two copies,

- `$GF_LIB_PATH/alltenses/`, the full library.
- `$GF_LIB_PATH/present/`, the library with a reduced tense system for verbs and sentences.

If the GF system is properly installed, one can see the library listing with the OS shell command

```
ls $GF_LIB_PATH/alltenses/
```

The two copies are produced from the same source files. The reason to provide a separate `present` variant is that, in many languages, the full tense system blows up the size of the grammar and makes compilation slower. For instance, in French, the total size of the binary RGL files is 3 MB in `alltenses` but just 1 MB in `present`. In many applications (the Phrasebook, technical language), only the present tenses are needed.

At the same time, advances in compiling GF are making the `present` version obsolete, and it is no longer supported for the most recent RGL languages. Using `alltenses` can still make compilation slower, but there is no run-time penalty with it, because unused tenses (like any unreachable word and phrase forms) can be eliminated by the **PGF optimization flag**,

```
gf -make -optimize-pgf FILES
```

The library sources are not necessary for programmers that just use the RGL and don't change it. However, they are included in the source distributions of GF, and they can also be viewed on-line in the subdirectories of

http://www.grammaticalframework.org/lib/src/

But perhaps the best view is obtained via the GitHub repository,

https://github.com/GrammaticalFramework/GF/tree/master/lib/src

with the usual GitHub navigation facilities and statistics of the latest changes.

4.4.3 The resource grammar Synopsis

The API level of RGL is documented in the **RGL Synopsis**,

http://www.grammaticalframework.org/lib/doc/synopsis.html

This document is automatically generated from the RGL sources on a regular basis. It is therefore up to date as regards to the current state of the library, *as available in the version control system*. This means that the previous standard release (package with a version number) might implement everything in the Synopsis. However, a Synopsis version corresponding to the release is included in the corresponding source package.

Navigating the Synopsis needs some experience. The best introduction is perhaps the GF book (Ranta 2011), Chapter 5—in particular, Section 5.15. Towards the beginning of the Synopsis, there is a tree-form chart (entitled "hierarchic view"), with clickable links to the most important categories. For instance, clicking at the circle "Cl" leads to the section on clauses, which has over 30 constructors whose value type is Cl. Each constructor is displayed with its type and an example in English. Hovering over the English example opens (for most functions) a list of translations of the example in all 25 languages. Figure 5 shows a snapshot of the Cl section of the Synopsis with translations.

The main part of the Synopsis corresponds to the module `Syntax`, which is implemented for all RGL languages. After that, the `Paradigms` API for each language is displayed. The documentation for the paradigms is less systematic and complete than for `Syntax`, because the languages don't always follow the same coding conventions. For instance, it is not always clear which forms should be given as arguments to the paradigms. A part of the German paradigms documentation is shown in Figure 4.4.3.

In addition to `Syntax` and `Paradigms`, the Synopsis shows the other modules on the API level. The internal modules are not shown, which partly reflects the fact that their use in applications is discouraged. But their source files can be viewed as expected, in which case their abstract syntaxes are the best API. For instance,

http://www.grammaticalframework.org/lib/src/german/ExtraGerAbs.gf

shows the abstract syntax of `ExtraGer`. Most of its contents are inherited from the common repository of possible extra functions,

http://www.grammaticalframework.org/lib/src/abstract/Extra.gf

The irregular verbs of German can be found in

http://www.grammaticalframework.org/lib/src/german/IrregGerAbs.gf

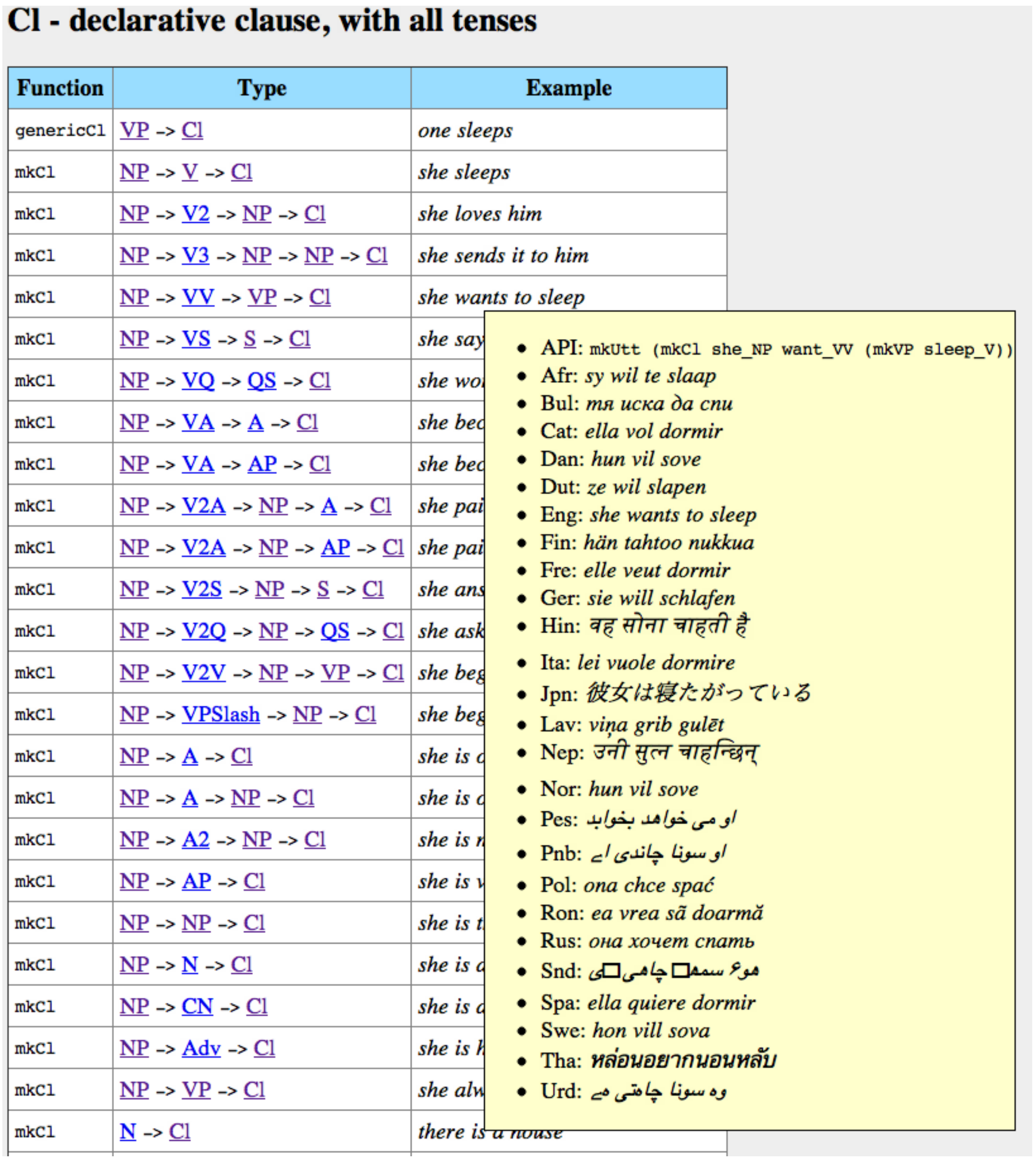
4.4.4 The resource grammar source browser

The RGL Source Browser,

http://www.grammaticalframework.org/lib/doc/browse/

gives an alternative, high-level view on the RGL sources of the internal layer. It has cross-links and a search facility. For each module, the "Scopes" view shows all identifiers in scope (also inherited ones), and the "Code" view show the actual code included in the module. The Scopes view moreover gives accurate links to the actual code. A snapshot is shown in Figure 4.4.4.

As the source browser discloses the internal layer, it is not a tool for browsing the library in standard application uses. But it can be useful for debugging grammars, because errors in application grammars may result from errors in the RGL.

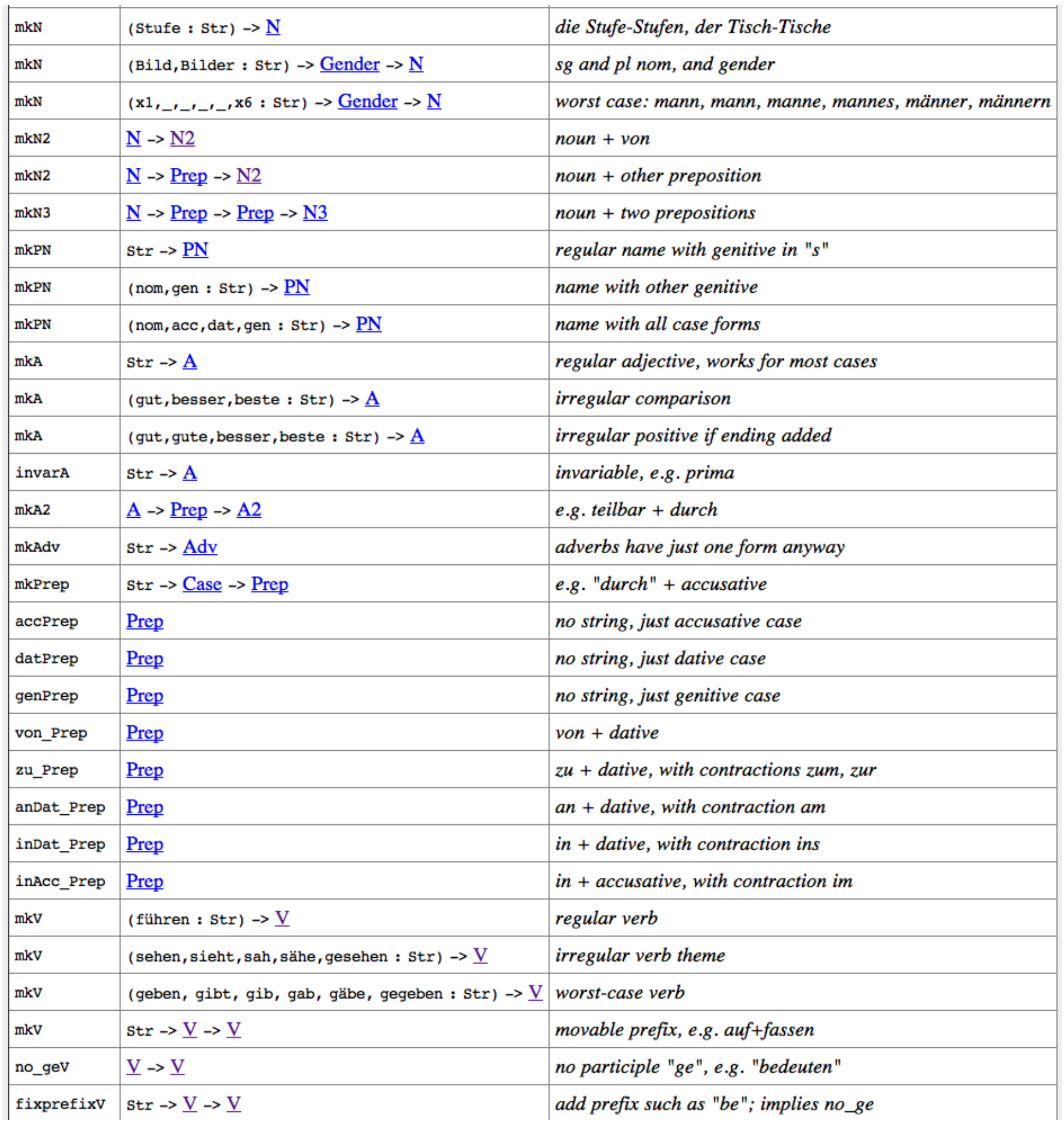


Cl - declarative clause, with all tenses

Function	Type	Example
`genericCl`	VP -> Cl	*one sleeps*
`mkCl`	NP -> V -> Cl	*she sleeps*
`mkCl`	NP -> V2 -> NP -> Cl	*she loves him*
`mkCl`	NP -> V3 -> NP -> NP -> Cl	*she sends it to him*
`mkCl`	NP -> VV -> VP -> Cl	*she wants to sleep*
`mkCl`	NP -> VS -> S -> Cl	*she say*
`mkCl`	NP -> VQ -> QS -> Cl	*she wo*
`mkCl`	NP -> VA -> A -> Cl	*she bec*
`mkCl`	NP -> VA -> AP -> Cl	*she bec*
`mkCl`	NP -> V2A -> NP -> A -> Cl	*she pai*
`mkCl`	NP -> V2A -> NP -> AP -> Cl	*she pai*
`mkCl`	NP -> V2S -> NP -> S -> Cl	*she ans*
`mkCl`	NP -> V2Q -> NP -> QS -> Cl	*she ask*
`mkCl`	NP -> V2V -> NP -> VP -> Cl	*she beg*
`mkCl`	NP -> VPSlash -> NP -> Cl	*she beg*
`mkCl`	NP -> A -> Cl	*she is o*
`mkCl`	NP -> A -> NP -> Cl	*she is o*
`mkCl`	NP -> A2 -> NP -> Cl	*she is m*
`mkCl`	NP -> AP -> Cl	*she is v*
`mkCl`	NP -> NP -> Cl	*she is t*
`mkCl`	NP -> N -> Cl	*she is a*
`mkCl`	NP -> CN -> Cl	*she is a*
`mkCl`	NP -> Adv -> Cl	*she is h*
`mkCl`	NP -> VP -> Cl	*she alw*
`mkCl`	N -> Cl	*there is a house*

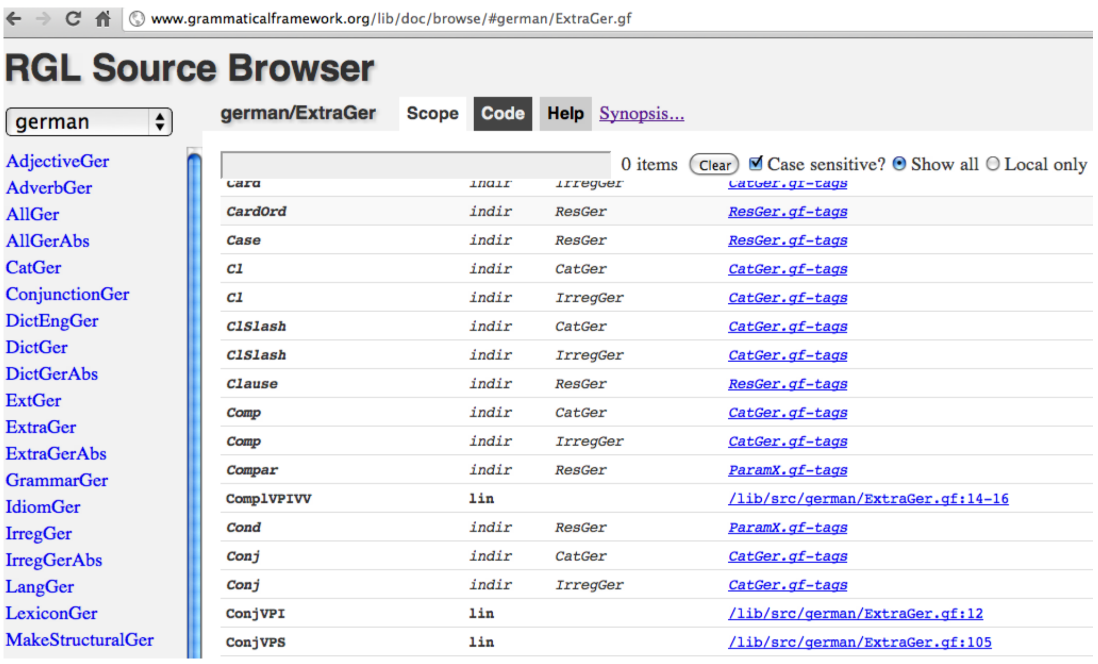
- API: `mkUtt (mkCl she_NP want_VV (mkVP sleep_V))`
- Afr: *sy wil te slaap*
- Bul: *тя иска да спи*
- Cat: *ella vol dormir*
- Dan: *hun vil sove*
- Dut: *ze wil slapen*
- Eng: *she wants to sleep*
- Fin: *hän tahtoo nukkua*
- Fre: *elle veut dormir*
- Ger: *sie will schlafen*
- Hin: *वह सोना चाहती है*
- Ita: *lei vuole dormire*
- Jpn: *彼女は寝たがっている*
- Lav: *viņa grib gulēt*
- Nep: *उनी सुत्न चाहन्छिन्*
- Nor: *hun vil sove*
- Pes: او می خواهد بخوابد
- Pnb: او سونا چاندی اے
- Pol: *ona chce spać*
- Ron: *ea vrea sã doarmă*
- Rus: *она хочет спать*
- Snd: هوء سمه□ چاهی□ی
- Spa: *ella quiere dormir*
- Swe: *hon vill sova*
- Tha: หล่อนอยากนอนหลับ
- Urd: وہ سونا چاھتی ہے

Figure 5: A part of the RGL Synopsis for clauses, with translations shown by hovering.



mkN	(Stufe : Str) -> N	*die Stufe-Stufen, der Tisch-Tische*
mkN	(Bild,Bilder : Str) -> Gender -> N	*sg and pl nom, and gender*
mkN	(x1,_,_,_,_,x6 : Str) -> Gender -> N	*worst case: mann, mann, manne, mannes, männer, männern*
mkN2	N -> N2	*noun + von*
mkN2	N -> Prep -> N2	*noun + other preposition*
mkN3	N -> Prep -> Prep -> N3	*noun + two prepositions*
mkPN	Str -> PN	*regular name with genitive in "s"*
mkPN	(nom,gen : Str) -> PN	*name with other genitive*
mkPN	(nom,acc,dat,gen : Str) -> PN	*name with all case forms*
mkA	Str -> A	*regular adjective, works for most cases*
mkA	(gut,besser,beste : Str) -> A	*irregular comparison*
mkA	(gut,gute,besser,beste : Str) -> A	*irregular positive if ending added*
invarA	Str -> A	*invariable, e.g. prima*
mkA2	A -> Prep -> A2	*e.g. teilbar + durch*
mkAdv	Str -> Adv	*adverbs have just one form anyway*
mkPrep	Str -> Case -> Prep	*e.g. "durch" + accusative*
accPrep	Prep	*no string, just accusative case*
datPrep	Prep	*no string, just dative case*
genPrep	Prep	*no string, just genitive case*
von_Prep	Prep	*von + dative*
zu_Prep	Prep	*zu + dative, with contractions zum, zur*
anDat_Prep	Prep	*an + dative, with contraction am*
inDat_Prep	Prep	*in + dative, with contraction ins*
inAcc_Prep	Prep	*in + accusative, with contraction im*
mkV	(führen : Str) -> V	*regular verb*
mkV	(sehen,sieht,sah,sähe,gesehen : Str) -> V	*irregular verb theme*
mkV	(geben, gibt, gib, gab, gäbe, gegeben : Str) -> V	*worst-case verb*
mkV	Str -> V -> V	*movable prefix, e.g. auf+fassen*
no_geV	V -> V	*no participle "ge", e.g. "bedeuten"*
fixprefixV	Str -> V -> V	*add prefix such as "be"; implies no_ge*

Figure 6: A part of the RGL Synopsis for German inflection paradigms.



Card	indir	IrregGer	CatGer.gf-tags
CardOrd	indir	ResGer	ResGer.gf-tags
Case	indir	ResGer	ResGer.gf-tags
Cl	indir	CatGer	CatGer.gf-tags
Cl	indir	IrregGer	CatGer.gf-tags
ClSlash	indir	CatGer	CatGer.gf-tags
ClSlash	indir	IrregGer	CatGer.gf-tags
Clause	indir	ResGer	ResGer.gf-tags
Comp	indir	CatGer	CatGer.gf-tags
Comp	indir	IrregGer	CatGer.gf-tags
Compar	indir	ResGer	ParamX.gf-tags
ComplVPIVV	lin		/lib/src/german/ExtraGer.gf:14-16
Cond	indir	ResGer	ParamX.gf-tags
Conj	indir	CatGer	CatGer.gf-tags
Conj	indir	IrregGer	CatGer.gf-tags
ConjVPI	lin		/lib/src/german/ExtraGer.gf:12
ConjVPS	lin		/lib/src/german/ExtraGer.gf:105

Figure 7: The RGL source browser for `ExtraGer`.

4.4.5 The resource grammar editor

RGL function combinations can become both long and deep. While the Synopsis helps finding functions and the GF compiler checks the type-correctness of their combinations, finding the proper combinations can be hard work. To help this, there is an experimental editor for building API function applications by using one of MOLTO's translation tools, the Fridge Magnet interface (the "Minibar"). It can be accessed in

http://cloud.grammaticalframework.org/minibar/minibar.html

where the proper choice of "Grammar" is "LibraryBrowser.pgf". A snapshot is shown in Figure 4.4.5.

The grammar editor supports input in English, Dutch, and RGL expressions. Users can create their own local versions with their own selections of languages by using a script included in

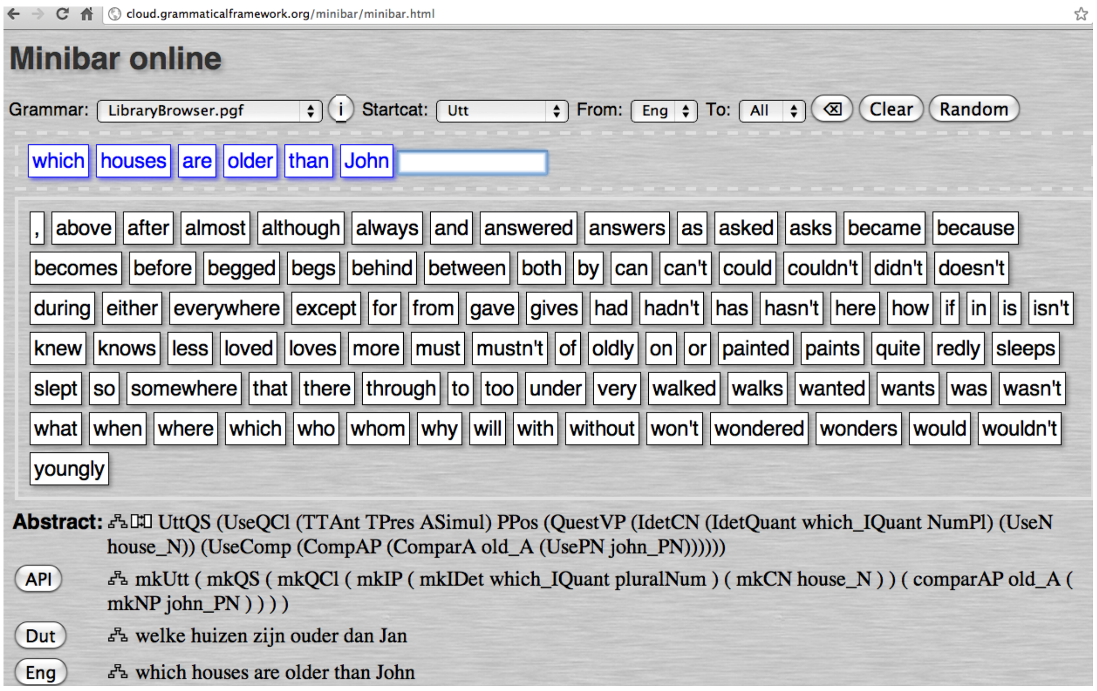
```
GF/lib/src/api/libraryBrowser/
```

in the GF sources. The editor is an application of the idea of **example-based grammar writing** (Section 6.3).

5 Writing a grammar

5.1 Creating a test corpus

Unlike most other grammar formalisms, GF uses separate descriptions of the abstract and the concrete syntax. Designing a grammar can therefore start from either of these. But a good starting point in either case is a set of examples, which cover the intended domain. From the concrete syntax point of view, these examples are sentences or other pieces of text. The usual practice is to start with English examples,


cloud.grammaticalframework.org/minibar/minibar.html
Minibar online
Grammar: LibraryBrowser.pgf
Startcat: Utt
From: Eng
To: All
Clear
Random
which houses are older than John
, above after almost although always and answered answers as asked asks became because
becomes before begged begs behind between both by can can't could couldn't didn't doesn't
during either everywhere except for from gave gives had hadn't has hasn't here how if in is isn't
knew knows less loved loves more must mustn't of oldly on or painted paints quite redly sleeps
slept so somewhere that there through to too under very walked walks wanted wants was wasn't
what when where which who whom why will with without won't wondered wonders would wouldn't
youngly
API
Dut
Eng

Abstract: UttQS (UseQCl (TTAnt TPres ASimul) PPos (QuestVP (IdetCN (IdetQuant which_IQuant NumPl) (UseN house_N)) (UseComp (CompAP (ComparA old_A (UsePN john_PN))))))

API mkUtt (mkQS (mkQCl (mkIP (mkIDet which_IQuant pluralNum) (mkCN house_N)) (comparAP old_A (mkNP john_PN)))))

Dut welke huizen zijn ouder dan Jan

Eng which houses are older than John

Figure 8: The RGL source browser for `ExtraGer`.

since most systems include English among their languages, and also because English is the language that is likely to be understood by later developers.

The test corpus can later be extended to a treebank usable for unit and regression testing (Section 4.3.3). The examples are also recommended as documentation of the abstract syntax: each `cat` and `fun` should be accompanied by a comment that gives an English example of the category or function, and perhaps also examples in other languages.

The simplest way to start with the test corpus is to collect some legacy texts, or to ask intended users to give examples of what they want to express in the language. It may turn out that some of the examples are not usable in the final grammar—for instance, they may be too ambiguous, ungrammatical, or not covered by the RGL. But one should always make sure to have at least some way to express every example by a suitable paraphrase, to make the grammar **semantically complete**.

5.2 Designing the abstract syntax

The abstract syntax can be partly determined by some existing formal system, for instance, if the goal of the grammar is to translate between the formal system and natural language. But usually it is better to abstract away from the exact details of the formalism. This is the experience from, for instance, the GF-KeY project (Johannisson 2005). That project created a software specification language based on the abstract syntax of OCL (Object Constraint Language). In later phases of the KeY project, OCL was replaced by JML (Java Modeling Language), and the grammars were no longer usable for the purpose of mapping between formal and informal specifications. The conclusion was that it would have been better to have a higher abstraction level in the abstract syntax, corresponding to "specifications in general". On the other hand, the OCL-based design had the undeniable advantage that the grammar was complete w.r.t. OCL.

Many applications of GF could be characterized as **description languages**. They are used for stating facts about some objects. They may also be used for posing questions, which makes them into **query languages**. But in general, one and the same grammar should support both assertions and questions. For instance, the simple grammar in Figure 2 does this. The support for both kinds of "speech acts" should be taken care of by a small number of high-level functions, such as `PQuestion` and `PAssertion` in Figure 2. The main part of the grammar is then devoted for building modality-neutral descriptions, for which Figure 2 uses the category `Statement`. Logically, these descriptions can be seen as **propositions**.

The best-known model for propositions is predicate calculus. This model includes connectives ("and", "or", "if", "not") and quantifiers ("all", "some"), as well as atomic propositions. Many descriptive languages are essentially weaker than full predicate calculus; for instance, OWL (Web Ontology Language), which is the target of many applications in MOLTO, is weaker. The intricacies of dealing with full predicate calculus are discussed in Ranta 2011b. In MOLTO, only the Mathematics case study (WP6) needs to deal with the expressive power of full predicate calculus. On the other hand, the Phrasebook application (Détrez et al. 2012) deals with content that has little to do with logic or descriptive language, such as greetings.

An application grammar may also extend beyond the level of single sentences, to cover **text** or even **discourse**. This is supported by the RGL, and exploited in MOLTO's mathematics and museum use cases. Writing grammars for entire texts can be an easy way to achieve cohesion in text, which is exemplified by the use of anaphora in the Museum case study (MOLTO Deliverable 8.2).

5.2.1 Predication

The basic types of a description language are **proposition** and **individual**. If we assume the categories

```
cat Prop  -- proposition
cat Ind   -- individual
```

we can define predicates as propositional functions,

```
fun Old  : Ind -> Prop          -- x is old
fun Like : Ind -> Ind -> Prop   -- x likes y
```

From propositions, we can in turn build assertions and questions, including negated ones. Defining predicates as functions is very powerful and general, because it allows the argument place to be just any position in the syntactic structure. For instance, if the predicate says that *the name of x is y*, we can use different structures in different languages:

English	`mkCl (mkNP (GenNP x) name_N) y`	*x's name is y*
German	`mkCl x (mkV2 heissen_V) y`	*x heisst y*
French	`mkCl x (mkV2 (reflV appeler_V)) y`	*x s'appelle y*

However, treating predicates as functions makes certain combinations impossible. The **modification** of a noun with a predicate cannot be expressed (*old man*). The **coordination** of two predicates cannot be expressed (*John is old and runs*). And the formation of **wh questions** becomes awkward, as it seems that one has to duplicate all predicates with variants that take interrogative phrases, rather than noun phrases, as arguments. Both of these problems can be solved by introducing special predicate categories:

```
cat Pred1   -- (x) is old
cat Pred2   -- (x) likes (y)
```

Then one can define applications to both noun phrases and interrogatives:

```
fun AppPred1Ind    : Pred1 -> Int    -> Prop   -- John is old
fun AppPred1Interr : Pred1 -> Interr -> Quest  -- who is old
```

The price to pay is that concrete syntax becomes more complex. We return to this in Section 5.3.

5.2.2 The module structure

For many applications, the natural structure is

base grammar + domain extension

The base grammar defines the high-level structures of assertions, questions, and complex propositions. It also defines most of the category structure. The domain extension defines, to use the predicate calculus terminology, the atomic propositions, which includes predicates and singular terms.

The rationale of the division is that the base grammar can be shared between many domain extensions. For instance, the MOLTO query language has a base grammar that is applied on three domains: patents, museum objects, and the Proton ontology of Ontotext.

Another characteristic of the division has to do with the complexity of the components. Thus the base grammar is often quite complex and needs more advanced grammarian skills, including knowledge of the linguistic concepts that structure the API. But the domain extensions are mainly lexical, and can be provided by layman informants, sometimes even by automatic transformations from ontologies (Section 6.1), lexical databases (Section 6.2), or statistical language models (Section 6.3).

5.2.3 Semantic control and dependent types

One of the fundamental questions in abstract syntax design is **semantic control**. In other words: should the grammar exclude meaningless expressions? The RGL has by design very little semantic control. Therefore it is equally meaningful to say

John likes beer

beer likes John

In abstract syntax terms, we have here a predicate that forms a proposition from two arguments:

```
fun Like : Ind -> Ind -> Prop
```

In the MOLTO phrasebook, this overgeneration is blocked by dividing noun phrases into more specific categories, such as `Person` and `Item`, and defining something equivalent to

```
fun Like : Person -> Item -> Prop
```

However, this differentiation has the potential of multiplying the number of rules in the grammar. For instance, one might also want to have

```
fun LikeP : Person -> Person -> Prop
```

linearized in the same way as `Like`.

A more elegant solution provided by GF is the use of **dependent types**. In the case at hand, this means having a category `Ind` parametrized by the category `Class`, which leads to the following definitions:

```
cat Class ; Ind (c : Class)
fun Person, Item : Class
fun Like : (c : Class) -> Ind Person -> Ind c -> Prop
```

This means that `Like` can only have a person as its subject, but a noun phrase of any class *c* as its object.

An abstract syntax without dependent types is called **context-free abstract syntax**. In terms of semantic control, we thus have three alternatives:

1. Simple, over-generating context-free abstract syntax.
2. Complex, restrictive context-free abstract syntax.
3. Restrictive abstract syntax with dependent types.

The third alternative is often rejected due to the known drawbacks of dependent types:

- It is a "difficult concept" not covered by basic GF (or programming) training.
- It is not supported by all run-time applications (e.g. the Java and C interpreters of PGF).
- It implies a risk of inefficiency, as the normal complexity limits don't apply.
- The targeted formal system performs its own semantic checking, so that the abstract syntax can be overgenerating.

The first reason, in particular, indicates that dependent types might not be recommendable in "best practices" for production use. But there is can still be good reasons to use them, and the museum case (MOLTO Deliverable 8.2) in fact gives an example of how they can be treated on a level hidden from "ordinary" grammar writers.

In practice, most application grammars apply some version of alternative (2)—complex, restrictive context-free abstract syntax. This is the case in the phrasebook, mathematics, museums, and query languages. However, the patent grammars and Attempto controlled language follow alternative (1). The largest ontology-based GF grammar, using the SUMO ontology, follows alternative (3) (Angelov and Enache 2012).

The last of the arguments against dependent types also applies to restrictive context-free grammars: it is redundant to perform semantic checking in the grammar, if some other component does it. However, the situation is not quite so simple. For instance, in GF-KeY (Johannisson 2005), dependent types were used in a syntax editor to guide the user to create only type-correct specifications. In dialogue systems, dependent types are used to reduce overgeneration, in order to define more restrictive language models and thereby better speech recognition performance (Jonson 2006).

Another interesting question to pose is whether semantic distinctions are relevant for translation. For instance, it may happen in some language that *x likes y* uses a different verb depending on whether *y* is a person, a drink, or something else. This is easy to handle if different predicates of different types are used, because they can be given different linearization rules:

```
lin Like  x y = Q x y
lin LikeP x y = P x y
```

But the whole idea of GF is that language-dependent distinctions need *not* be reflected in the abstract syntax. For instance, the hypothetical translations of `Like` can also be captured by an inherent feature attached to NP's:

```
param Class = Person | Item ...
lincat Ind = {np : NP ; c : Class}
lin Like x y = case y.c of {Person => P x.np y.np ; _ => Q x.np y.np}
```

This solution scales up better than abstract syntax distinctions, because it is stable when new languages are added.

5.3 Mapping to resource grammar categories

Ideally, the abstract syntax is designed independently of the concrete syntax. The next step is to give the `lincat` rules that map abstract categories to RGL types. Among the 83 categories of the RGL, only a handfull are recommended as linearization types in application grammars:

`Text`	texts, punctuated sentences
`Utt`	top-level utterances (if not top-level)
`S`	declarative sentences with fixed tense and polarity
`QS`	questions with fixed tense and polarity
`Cl`	clauses (predications) with variable tense and polarity
`CN`	common nouns: types, classes, kinds
`NP`	noun phrases: names, subjects, objects
`AP`	adjectival phrases: properties, qualities
`Adv`	adverbs, prepositional phrases
`Card`	cardinal numbers - either symbolic or verbal

Other categories should be avoided, for different reasons. One useful rule to observe is that

The chosen linearization types should work in all languages.

This makes it possible to use **functors** in concrete syntax, if wanted, and in any case makes concrete syntax implementations more portable. The main classes of categories excludes are

- **lexical categories**, such as `N`, `A`, `V`, because it *very* often happens that e.g. a lexical noun (`N`) in one language has to be rendered as a complex noun (`CN`) in another language.
- **structural word categories**, such as `Det`, `Quant`, `RP`, which are best handled by functions over base categories instead.
- **theory-laden categories**, such as `VP`, `VPSlash`, which can be difficult for other application grammarians to understand.

Sometimes, however, a theory-laden category solves a problem by introducing more generality. For instance, the abstract category of one-place predicates may sometimes be expressed by an adjective, sometimes by a verb. Using `VP` as `lincat` will cover both cases. Likewise, `VPSlash` can be used for two-place predicates.

5.3.1 Records of categories

Simple RGL categories don't always capture all uses of abstract syntax elements. For instance, in the MOLTO Phrasebook, the category `Nationality` is used in three different ways:

```
LangNat     : Nationality -> Language    -- Swedish
CitiNat     : Nationality -> Citizenship -- Swedish
CountryNat : Nationality -> Country     -- Sweden
```

These ways require three kinds of expressions to be derivable from a `Nationality`: a language name, an adjective, and a country name. The natural linearization type for this is a **record type** with all these parts as components:

```
lincat Nationality = NPNationality ;
oper NPNationality : Type = {lang : NP ; country : NP ; prop : A}
```

It is usually a good idea to introduce a **type synonym** (here `NPNationality`) to refer to such complex types, so that the code will be easier to change if needed. For instance, in French it would turn out that the type needs a fourth component, giving the preposition that is used for saying that someone is in the country (*en France* as opposed to *au Danemark*):

```
oper NPNationality : Type =
  {lang : NP ; country : NP ; prop : A ; prep : Prep}
```

For the same reason, actual linearization rules for nationalities should use **constructors** rather than explicit records. The naming conventions for constructors in the RGL is recommended also in application grammars: use overloaded constructors with the name `mk`*C*, covering both the worst case and the most common "regular" cases. For instance, English nationality terms can be built with the following constructors:

```
oper mkNationality = overload {
  mkNationality : (adj,noun : Str) -> NPNationality = \adj,noun ->
    {lang = mkNP (mkPN adj), country = mkNP (mkPN noun), prop = mkA adj} ;
  mkNationality : (lang,cou : NP) -> (prop : A) -> NPNationality =
    \lang,cou,prop -> {lang = lang ; country = cou ; prop = prop} ;
 }
```

With these constructors, the lexicon builder can easily add both regular and irregular cases:

```
lin Danish = mkNationality "Danish" "Denmark"
lin Dutch = mkNationality (mkNP (mkPN "Dutch"))
              (mkNP the_Det (mkN "Netherlands")) (mkA "Dutch")
```

The constructors should be defined in the base module (Section 5.2.2), so that all low-level details about the types are hidden from the domain extensions.

5.4 Porting a grammar to a new language

The recommended workflow for porting a grammar to a new language is as follows. Assume we are porting English (Eng) to German (Ger).

1. Copy the Eng files to corresponding Ger files.
2. Replace all references in the module header from Eng modules to Ger modules.

These steps are completely mechanical. With good luck, they may result in a compilable German grammar, which can be tested in GF. "Good luck" means that the Eng files use no `Extra`, `Irreg`, or `Paradigms` functions, which are not defined for German. If successful, this grammar will generate German syntax with German structural words and English content words, the latter with German morphology. This grammar is probably far from acceptable, but can (with the "good luck") be produced in a few minutes. For example, the German Shops grammar (Figure 2) obtained in this way will generate sentences like

Ist der bar closed? Ist der shop open? Ist die station open?

Der bar ist closed. Der shop ist open. Die station ist open.

Some more work is therefore needed:

3. Change the strings in the module from English to German words.

xThe resulting sentences look more German:

Ist der Bar geschlossen? Ist der Geschäft geöffnet? Ist der Bahnhof geöffnet?

Der Bar ist geschlossen. Der Geschäft ist geöffnet. Der Bahnhof ist geöffnet.

Almost everything is correct now—except the genders of the nouns. Therefore:

4. Add more information to the lexical paradigm applications if necessary.

In the present case, this also requires the addition of new constructors: the English grammar assumed only one string is needed to define a place name, but this doesn't work in German.

5. Add more cases to constructors.

Here we only need to add one constructor for `mkPlace`, taking nouns rather than strings as arguments:

```
mkPlace : N -> NP = \n -> mkNP the_Det n
```

Then we can define

```
Bar = mkPlace (mkN "Bar" "Bars" feminine)
Shop = mkPlace (mkN "Geschäft" "Geschäfte" neuter)
```

The test suite now comes out completely correct:

Ist die Bar geschlossen? Ist das Geschäft geöffnet? Ist der Bahnhof geöffnet?

Die Bar ist geschlossen. Das Geschäft ist geöffnet. Der Bahnhof ist geöffnet.

The Shops grammar does not require the most radical changes:

6. Change the applications of Syntax API functions if needed.

7. Change linearization types if needed, and the affected constructors and linearization rules accordingly.

If none of these are needed, the translation works without compile-time transfer (Section 2.3), i.e. by a "literal translation" using the same syntactic structures as in the old languages. In this situation, porting the grammar requires very few skills—basically, just knowledge of words, not of syntactic structures.

5.4.1 Using a functor

Literal translation makes it possible to use a **functor**, that is, a module parametrized on the RGL Syntax interface and possibly a lexicon interface specific to the application. The GF book (Ranta 2011, sections 5.8-5.13) explains the use of functors and also how they can be overridden.

The advantages of functors are obvious:

- Less source code is needed.
- Porting a grammar to a new languages needs just the minimum of work.
- If the abstract syntax is changed, concrete syntax needs to be changed in just one place.

Their disadvantages are also important to know:

- The concept of a functor is advanced and not previously known to many programmers.
- Debugging functorized code can be hard due to its many levels.
- Compile-time transfer is more difficult than without functors.

The first disadvantage, in particular, makes it questionable if functors belong to the "best practices". They will be familiar for ML and OCaml programmers (not very common), and perhaps also to C++ programmers because of their resemblance of templates. On the other hand, the unfamiliarity can be relieved if functors are limited to the base grammars, which are build by more experienced GF programmers; the bulk of most grammars belongs to domain extensions, which don't use functors—and wouldn't benefit a lot from them either, since they are mostly lexical.

6 Automatic grammar generation

In a typical application grammar, the base grammar is a few dozen rules, and the rest is domain extension Section 5.2.2. The combination rules of the base grammar use syntactic structures and hence need skill and knowledge, whereas large parts of the domain extension can be automated. Of course, if 100% precision is expected from the grammar, the results of automatic generation have to be inspected manually; but this is usually much faster than writing everything manually.

6.1 From ontologies

MOLTO uses ontologies in different formats: while many tasks use OWL (e.g. Dannélls et al. 2012), grammar generation has been performed on SUMO (Angelov and Enache 2012) and Linked Open Data (Davis et al. 2012). The goal is to create general methods for mapping ontologies to grammars, rather than focus on specific formats; case studies on different formats are of course essential to demonstrate that the methods actually work in practice.

The main methods of grammar generation are derived from general principles of description languages, ultimately from formal logic (Section 5.2). Most concepts present in ontologies can be classified to a system that comprises the following categories:

- **types**, also known as categories, domains, sets, or classes, e.g. *customer*
- **individuals**, also known as elements, instances, terms, or entities, e.g. *Morgan Stanley*

- **propositions**, also known as facts, descriptions, formulas, or statements, e.g. *YouTube is owned by Google*
- **predicates**, also known as properties, qualities, or propositional functions, e.g. *is active*
- **relations**, also known as 2-place predicates, e.g. *is owned by*
- **functions**, assigning individuals to individuals e.g. *the owner of*

Some general decisions have to be made in the base grammar:

1. Whether types are formalized as categories or functions.

This decision is directly dependent on the amount of **semantic control** wanted (Section 5.2.1). With weak semantic control, types are formalized as functions, and there is just a general category of individuals. With strong semantic control, the types of the ontology enter into the category system, either as separate categories or as argument to dependent types. It should be notice that a grammar with strong semantic control also has to deal with **subtyping**, for which dependent types are a more efficient tool than context-free categories (Angelov and Enache 2012).

2. Whether predicates and relations are formalized as functions or by specific categories.

This question was discussed in Section Section 5.2.3. The conclusion was that the use of functions gives more freedom in intra- and cross-lingual variation, but the use of specific categories gives more combination possibilities due to for instance coordination and Wh question formation. It is also possible to divide the predicate category to subcategories, corresponding for instance to adjectives and verbs (with `AP` and `VP` as corresponding linearization type). This idea is followed in the Attempto Controlled Language grammar (MOLTO WP 11). It may sometimes fail to scale up to new languages, because expressing a concept as an adjective or as a verb is not invariant.

Answers to these questions lead to the following baseline formalizatins of ontological concepts. We distinguish between the three choices of semantic control described in Section 5.2.3.
1. With weak control:

concept	abstract syntax	concrete syntax
types *T*	`cat Typ, fun T : Typ`	`lincat Typ = CN`
individuals *i*	`cat Ind, fun i :  Ind`	`lincat Ind = NP`
propositions *A*	`cat Prop, fun A : Prop`	`lincat Prop = Cl`
predicates *P* as functions	`fun P : Ind -> Prop`	`lin P x = mkCl ...`
predicates *P* as category	`cat Pred ; fun P : Pred`	`lincat Pred = VP`

2. With strong control, context-free abstract syntax:

concept	abstract syntax	concrete syntax
types *T*	`cat T`	`lincat T = CN`
individuals *i*	`fun i :  T`	`lin i = mkNP ...`
propositions *A*	`cat Prop, fun A : Prop`	`lincat Prop = Cl`
predicates *P* as functions	`fun P : T -> Prop`	`lin P x = mkCl ...`
predicates *P* as category	`cat PredT ; fun P : PredT`	`lincat PredT = VP`

3. With strong control, dependent types:

concept	abstract syntax	concrete syntax
types *T*	`cat Typ, Ind Typ`	`lincat Typ = CN, Ind = NP`
individuals *i*	`fun i :  Ind T`	`lin i = mkNP ...`
propositions *A*	`cat Prop, fun A : Prop`	`lincat Prop = Cl`
predicates *P* as functions	`fun P : Ind T -> Prop`	`lin P x = mkCl ...`
predicates *P* as category	`cat Pred Typ ; fun P : Pred T`	`lincat Pred = VP`

Of course, the names of the categories can be chosen to reflect the ontology.

The next step is to map ontology concepts to GF rules. For each concept, an abstract syntax function and its linearization rule is generated. This can be made mechanically by using **smart paradigms** with their regular (one-argument) variants (see Section 6.2). For instance, assuming the choice of Alternative 1, weak semantic control, we can for instance generate

type *Customer*:

```
fun Customer : Typ
lin Customer = mkCN (mkN "customer")
```

individual *MorganStanley*:

```
fun MorganStanley : Ind
lin MorganStanley = mkNP (mkPN "MorganStanley")``
```

predicate *IsActive*:

```
fun IsActive : Ind -> Prop
lin IsActive x = mkCl x (mkA "active")
```

predicate *IsOwnedBy*:

```
fun IsOwnedBy : Ind -> Ind -> Prop
lin IsOwnedBy x y = mkCl x (mkVP (passiveVP (mkV2 "own") y))
```

Notice that we here exploit the **camel script** notation of ontology names to infer the structure of terms—the compound structure of *MorganStanley* and the syntactic structures of *IsActive* and *IsOwnedBy*. Inferring the syntactic structure in this way belongs to the realm of **example-based grammar writing**, which is discussed in Section 6.3.

Converting ontologies to the other types of abstract syntax (Alternatives 2 and 3) is essentially similar to Alternative 1, since the linguistic information is independent of semantic control.

The method described above works for ontology-to-grammar conversion for one language. The next step is to port it to other languages. The manual procedure is discussed in Section 5.4. Some automation can be achieved by using terminologies and multilingual lexica.

Lexicon	**Forms**	**Entries**	**Cost**	$m = 1$	$m \leq 2$
Eng Noun	2	15,029	1.05	95%	100%
Eng Verb	5	5,692	1.21	84%	95%
Swe Noun	9	59,225	1.70	46%	92%
Swe Verb	20	4,789	1.13	97%	97%
Fre Noun	3	42,390	1.25	76%	99%
Fre Verb	51	6,851	1.27	92%	94%
Fin Noun	34	25,365	1.26	87%	97%
Fin Verb	102	10,355	1.09	96%	99%

Table 3: The predictive power of smart paradigms in four languages.

6.2 From lexical databases

Lexical databases can be used for two purposes:

- monolingual information: inflection, gender, valency, etc
- multilingual links

Thus, instead of the use of smart paradigms in Section 6.1, one could use a trusted monolingual lexicon, such as the `Dict` modules of the RGL. Table 1 shows the currently available `Dict` modules.

However, ontologies often contain very special technical terminology, which is not covered by general-purpose dictionaries. The use of the `Paradigms` modules of the RGL is therefore often necessary. Most languages (as indicated in Table 3) have a system of **smart paradigms**, which often gives very good predictions of full inflection tables from just one or two forms. The method is described and in Détrez and Ranta 2012, and also evaluated for four languages on large trusted dictionaries. The main results are in Table 6.2 (from Détrez and Ranta 2012). The $m = 1$ column indicates the percentage of words whose inflection is correctly inferred from just one form. Since this is usually the dictionary form, and the one used as an identifier in ontologies and other word lists, we can get a reasonable approximation in all categories above, except Swedish and French nouns (where the main problem is to infer the gender). The "Cost" column indicates the average number of forms needed, and therefore an estimate for the amount of manual work needed.

For interlingual links, the main sources tried out in MOLTO are WordNet, Wiktionary, and Wikipedia. Wikipedia gives a good precision for technical terms, but a low coverage. Linked WordNets give a good coverage, but the problem is ambiguity: any single word is typically linked to many alternatives, and it can be impossible to choose the correct one with available information. One thing that can help is contextual information, which leads us to the method of example-based grammar generation.

6.3 From examples

Example-based grammar writing means the extraction of linearization rules from examples. In this method, an example of a GF rule (for instance, a predicate) is first presented to an oracle, which can be a native speaker informant but also an SMT system. The obtained translation is parsed in the RGL. The parse tree is generalized with respect to relevant variables to obtain the desired linearization rule.

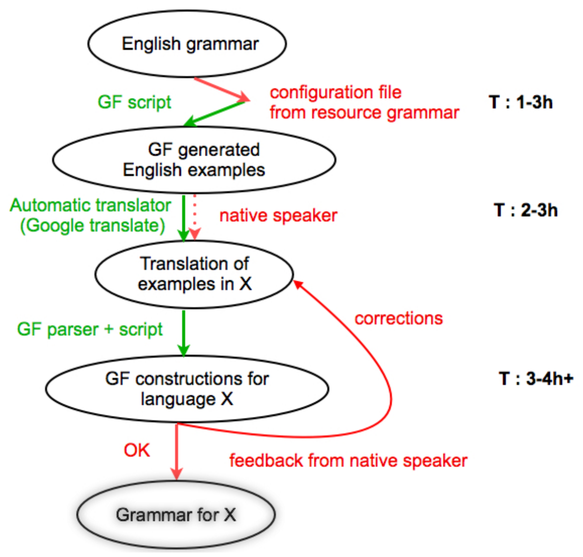

English grammar
GF script
configuration file from resource grammar
T : 1-3h
GF generated English examples
Automatic translator (Google translate)
native speaker
T : 2-3h
Translation of examples in X
corrections
GF parser + script
GF constructions for language X
T : 3-4h+
OK
feedback from native speaker
Grammar for X

Figure 9: The work flow in example-based generation of MOLTO Phrasebook modules.

Abstract syntax	`Like She He`	first grammarian
English example	*she likes him*	first grammarian
German translation	*er gefällt ihr*	native speaker/SMT system
resource tree	`mkCl he_NP gefallen_V2 she_NP`	GF parser
concrete syntax rule	`lin Like x y = mkCl y gefallen_V2 x`	variables renamed

The work flow as tested in the MOLTO Phrasebook is described in Figure 6.3 from Détrez et al. 2012 (see also Appendix C).

Using native informants is a natural idea, whereas the use of SMT can be cheaper, and can be surprisingly good. The reason is that:

- SMT is *good* with short, common sentences
- SMT is *bad* for sentences that are *long* and involve *word order variations*
- GF can generalize from one example to all combinations

For instance, the following kind of sentences can then be translated correctly:

if you like me, I like you

`If (Like You I) (Like I You)`

wenn ich dir gefalle, gefällst du mir

Pure SMT has problems with the other word orders, but as long as it gets main clauses right, the other ones are generated by grammar-based reordering.

The resource grammar editor (Section 4.4.5) can be seen as an implementation of the example-based method. It is interactive rather than automatic. One of the still unsolved problems in automating the method is the disambiguation of examples that give rise to several RGL trees. For example, the prepositional phrase *by Google* in *YouTube is owned by Google* can be parsed as the agent of a passive construction, but also as a locative (as in *by the river*) or a temporal (as in *by the midnight*). Choosing the correct tree is not crucial in English, but naively porting the resulting grammar to other languages can give strange results.

7 Best practices: a summary

To make your work reusable, and to enable a division of labour:

- *Divide the grammar into a base module (syntactic) and domain extension (lexical).*

To make it maximally simple to add languages:

- *Consider defining the base part by a functor.*

To avoid low-level hacking and guarantee grammatical correctness:

- *In the concrete syntax, use only function applications and string tokens, maybe records - but no tables, no concatenation.*

To guarantee that the grammar will continue to work in the future:

- *Only use the API level of the resource grammar library.*

For scalability:

- *Choose solutions that remain stable when new languages are added.*

A corollary:

- *Never use lexical categories as linearization types.*

A scalability tools:

- *Use type synonyms and constructors rather than raw types for linearization.*

To make the grammar compilable by others:

- *All paths must be in terms of standard library relative to GF_LIB_PATH and other directories in the same project - no private directories.*

To monitor your progress:

- *Create a treebank for unit and regression testing, and use it often with the diagnostic tools.*

The following tools are standard and well-tested in MOLTO's and other applications:

- the GF compiler and shell
- the GF run-time for Haskell, Java, and C, as well as the web API

- the RGL for the 15 MOLTO languages
- the GF-Eclipse IDE
- the use of smart paradigms for lexicon building

The following tools are actively developed, but still more research prototypes than production tools:

- the RGL for the other languages
- the cloud-based IDE for GF
- example-based grammar writing
- ambiguity testing and other automated formal grammar methods

References

M. Ahlberg and R. Enache. A Type-Theoretical Wide-Coverage Computational Grammar for Swedish. *TSD 2012*, to appear. 2012.

K. Angelov. *The Mechanics of the Grammatical Framework*. PhD Thesis, Chalmers University of Technology, 2012.

K. Angelov and R. Enache. Typeful Ontologies with Direct Multilingual Verbalization. N. Fuchs and M. Rosner (eds), *CNL 2010 proceedings*, Springer LNCS/LNAI vol. 7175. 2012.
http://www.molto-project.eu/sites/default/files/FinalSUMOCNL.pdf

Y. Bar-Hillel. *Language and Information*, Addison-Wesley, Reading, Ma., 1964.

B. Bringert, R. Cooper, P. Ljunglöf, A. Ranta. Multimodal Dialogue System Grammars. *Proceedings of DIALOR'05, Ninth Workshop on the Semantics and Pragmatics of Dialogue, Nancy, France, June 9-11, 2005*, 2005.

J. Camilleri. An IDE for the Grammatical Framework, *Proceedings of FreeRBMT12*, to appear, 2012.

D. Dannélls, R. Enache, M. Damova, and M. Chechev. Multilingual Online Generation from Semantic Web Ontologies. *WWW2012, 04/2012, Lyon, France*, (2012).
http://www.molto-project.eu/sites/default/files/wwweu2012_submission_16-final.pdf

B. Davis, R. Enache, J. van Grondelle and L. Pretorius. Multilingual Verbalisation of Modular Ontologies using GF and lemon. *CNL 2012*, to appear, 2012.
http://www.molto-project.eu/sites/default/files/Multilingual%20Verbalization%20of%20Modular%20Ontologies%20usin

G. Détrez and A. Ranta. Smart Paradigms and the Predictability and Complexity of Inflectional Morphology. *EACL (European Association for Computational Linguistics)*, Avignon, April 2012.

G. Détrez, R. Enache, and A. Ranta. Controlled Language for Everyday Use: the MOLTO Phrasebook. N. Fuchs and M. Rosner (eds), *CNL 2010 proceedings*, Springer LNCS/LNAI vol. 7175.

M. Forcada, M. Ginesti-Rosell, J. Nordfalk, J. O'Regan, S. Ortiz-Rojas, J. Perez-Ortiz, F. Sanchez-Martinez, G. Ramirez-Sanchez and F. Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation* 24, pp. 1–18. 2011.

K. Johannisson. *Formal and Informal Software Specifications*. PhD thesis, Computer Science, University of Gothenburg, 2005.

R. Jonson. Generating statistical language models from interpretation grammars in dialogue system. *Proceedings of EACL'06*, Trento, Italy. 2006.

P. Koehn. *Statistical Machine Translation*. Cambridge University Press. 2010.

A. Ranta. *Grammatical Framework: Programming with Multilingual Grammars*, CSLI Publications, Stanford, 2011.

A. Ranta. Translating between Language and Logic: What Is Easy and What is Difficult? In N. Bjørner and V. Sofronie-Stokkermans (eds.), *CADE-23. Automated Deduction*, LNCS/LNAI 6803, pp. 5-25, 2011b.

A Cloud-Based IDE Manual: Abstract

This document specifies the functionalities for an IDE (Integrated Development Environment) for GF (Grammatical Framework): how should such a system help the programmers who write multilingual grammars? Two IDE's are presented: a web-based IDE enabling a quick start for GF programming in the cloud, and an Eclips plug-in, targeted for expert users working with large projects, which may involve the integration of GF with other components. As an emerging technology, the document concludes with example-based grammar writing, which is a functionality that enables the use of natural language examples rather than GF code in grammar writing.

Full text: A. Ranta, T. Hallgren, et al. *Grammar IDE*, MOLTO Deliverable D2.2, September 2011.
http://www.molto-project.eu/sites/default/files/http:www.molto-project.eu:print:book:export:html:1379.pdf

B GF-Eclipse Manual: Abstract

The GF Eclipse Plugin provides an integrated development environment (IDE) for developing grammars in the Grammatical Framework (GF). Built on top of the Eclipse Platform, it aids grammar writing by providing instant syntax checking, semantic warnings and cross-reference resolution. Inline documentation and a library browser facilitate the use of existing resource librari es, and compilation and testing of grammars is greatly improved through single-click launc h configurations and an in-built test case manager for running treebank regression tests. This IDE promotes grammar-based systems by making the tasks of writing grammars and using resource libraries more efficient, and provides powerful tools to reduce the barrier to en try to GF and encourage new users of the framework.

Full text: J. Camilleri. An IDE for the Grammatical Framework, *Proceedings of FreeRBMT12*, to appear, 2012.
http://www.molto-project.eu/sites/default/files/freerbmt2012.pdf

C Controlled Language for Everyday Use: Abstract

Controlled languages are usually targeted for technical domains and designed to be unambiguous. This paper presents a controlled language whose domain is touristic phrases, aimed to be usable by anyone without prior training. Despite its informal nature, the language of phrases has a firm notion of semantics, defining the correctness of translations. However, this semantics is formulated in terms of context and situation rather than by logical formulas. Moreover, the language is often ambiguous, and the translation may depend on resolving the ambiguity. This paper shows how to formalize a semantics for tourist phrases and implement it in 15 languages, how to deal with the ambiguities, and how to make the system available for layman users on the web and on mobile phones. While a useful application as such, the Phrasebook also paves the way for an extended notion of controlled language, and the techniques are aimed to be general enough to support many such extensions.

Full text: G. Détrez, R. Enache, and A. Ranta. Controlled Language for Everyday Use: the MOLTO Phrasebook. Fuchs & Rosner (eds), *CNL 2010 proceedings*, Springer LNCS/LNAI vol. 7175.
http://www.molto-project.eu/sites/default/files/everyday.pdf

Index