Best Practices in GF Grammar Writing

Aarne Ranta

GF Summer School 2013, based on MOLTO Deliverable 2.3

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	A. Ranta, J. Camilleri, G. Détrez, R. Enache, T. Hallgren
Task responsible	UGOT
Other contributors	O. Caprotti, D. Dannélls, I. Listenmaa, J. Saludes

http://www.molto-project.eu/sites/default/files/MOLTO_D2.3.pdf

MOLTO's mission

Producer-oriented translation: providers of information can rely on the translations so much that they can publish them.

As opposed to **consumer-oriented** translation, which is applied by consumers to get a rough idea of what the original document is about.

- This is the main scenario in current machine translation (e.g. Google translate, Microsoft Bing)

Dissemination = producers' translation

Assimilation = consumers' translation

Responsibility

Think of a French e-commerce site text

prix: 99 euros

by accident translated to Swedish

pris: 99 kronor

(99 kronor = 11 EUR)

Is the e-commerce site committed to the price?

- yes, if it is producer's transtion
- no, if it is consumer's translation

FAHQT

Fully Automatic High-Quality Translation, “impossible, not only in foreseeable future, but in principle” (Bar-Hillel 1964)

Bar-Hillel’s example:

> *The pen is in the box.*
>
> *The box is in the pen.*

How to translate *pen* into Swedish? In one sense, *pen* = *penna* (writing utensil). In another sense, *pen* = *lekhage* (enclosure where children play).

To select the sense requires unlimited intelligence and a “universal encyclopedia”.

Solution: restricted language

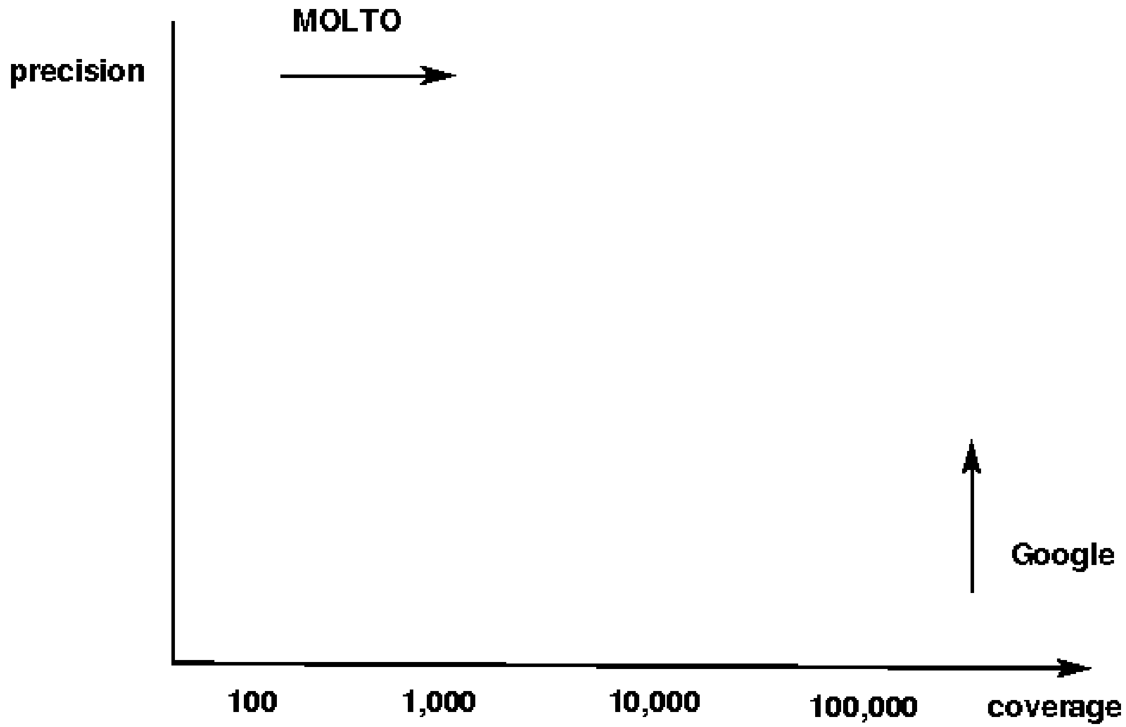
Consumer tools must cope with any document that is thrown at them.

Producer tools can assume a limited fragment of language.

Thus an e-commerce site can be restricted to translating product descriptions, order forms, and customer agreements.

MOLTO's mission was to make this feasible for different users and scenarios, and scalable from small fragments of few languages to larger fragments of large numbers of simultaneous languages.

Approaching full precision and full coverage



x-axis: the number of “concepts” (words, multi-word terms, constructions)

How to make restricted FAHQT feasible

The GF programming language and compiler

The Resource Grammar Library

Tools for grammar writers

Tools for translation

Building a web-based translation system

Two steps:

1. Write a multilingual grammar.

2. Set up a web interface.

The first step requires manual programming work.

The second step can use GF's standard interfaces out of the box, or modify them to different purposes.

A quick example: shops

A small fragment of the MOLTO phrasebook (Détrez et al., 2012).

Phrases like *the bar is open*

English, Finnish, and French.

The complete grammar code

```
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 ;
}
```

```
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) ;
}
```

```
concrete ShopsFin of Shops = open SyntaxFin, ParadigmsFin in {

flags coding = utf8 ;

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 ;
}
```

```
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) ;
}
```

The translation system in action

1. Select a property.

2. Obtain translations.

3. Change “bar” to “station”.

Building a web application

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/
```

4. The translator can now be accessed in `http://localhost:41296/`

Linguistic knowledge from the RGL

Even these most trivial natural language grammars involve expert linguistic knowledge.

Word inflection: French *ouvert-ouverte*

Gender agreement: French *le bar est ouvert* ("the bar is open", masculine) vs. *la gare est ouverte* ("the station is open", feminine)

The grammar code `ShopsFre.gf` does not mention gender or agreement.

And it has no occurrences of the words *la*, *le*, *ouverte*!

Language differences and the RGL

Automatic in the grammar: renderings of common operations

- Example: the definite article `the_Def`
  - English: one form *the*
  - French: several forms *le, la, l', les*
  - Finnish: no word at all

Decided by the grammarian: choices of operations

- Example: the predicates “open” and “closed”
  - English and French: adjectives
  - Finnish: adverbs

Questions for best practices

- *When should I use GF rather than something else?*
- *What domains can the translations address?*
- *What applications and use cases are there?*
- *What languages have been dealt with?*
- *How much time does it take to build a translation system?*

- *What skills and training are needed?*

- *What alternative tools are there?*

Indications for using GF

Aim at full precision

Maximally thousands of concepts

A high number of languages

Presence of morphologically complex languages

Availability of RGL

MOLTO applications

Phrasebook: touristic phrases

Math: mathematical concepts from OpenMath

Museum: museum object descriptions and queries, mostly from DbPedia

ACE: Attempto Controlled English, based on predicate logic and OWL

Patent: legacy patent texts from the pharmaceutical domain

Languages used in MOLTO applications

Language	Code	Smart	Dict	MOLTO	Phraseb	Math	Museum	ACE	Patent
Bulgarian	Bul		+	+	+	+	+	+	
Catalan	Cat	+		+	+	+	+	+	
Chinese	Chi	+						+	
Danish	Dan	+		+	+		+	+	
Dutch	Dut	+		+	+		+	+	
English	Eng	+	+	+	+	+	+	+	+
Finnish	Fin	+	+	+	+	+	+	+	
French	Fre	+	+	+	+	+	+	+	+
German	Ger	+	+	+	+	+	+	+	+
Greek	Gre	+						+	
Hebrew	Heb						+		
Hindi	Hin	+	+		+			+	
Italian	Ita	+		+	+	+	+	+	
Latvian	Lav	+			+			+	
Norwegian	Nor	+		+	+		+	+	
Persian	Pes				+				
Polish	Pol			+	+	+		+	
Romanian	Ron	+		+	+	+	+	+	
Russian	Rus	+	+	+	+	+	+	+	
Spanish	Spa	+		+	+	+	+	+	
Swedish	Swe	+	+	+	+	+	+	+	
Thai	Tha	+			+			+	
Urdu	Urd	+			+	+		+	
total	23	19	8	15	20	13	15	21	3

RGL but no MOLTO applications: Afrikaans, Japanese, Maltese, Nepali, Punjabi, Sindhi

Development effort for the MOLTO Phrasebook

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	##	###	-	+	-	-	##

Fluency: ### = native. **GF skills**: # = two days' tutorial. **Effort**: # = 1 working day

Alternative tools

Consumer applications

- Statistical Machine Translation (SMT)
- Apertium (rule-based open-domain shallow-transfer system)

Producer applications

- HPSG (Head-Driven Phrase Structure Grammar), Lingo Matrix grammars, and LKP tool

- LFG (Lexical-Functional Grammar), ParGram grammars, and XLE tool

- Regulus, a Prolog-based system specialized for spoken language translators

The choice of tools

All available from http://grammaticalframework.org under open-source licenses

The GF grammar compiler

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

GF IDE's

(Integrated Development Environments)

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

Grammar diagnostic tools

- displaying grammar information
- statistics about a grammar
- ambiguity checking
- unit and regression testing

The GF Resource Grammar Library

Morphology, syntax, and lexicon for 28 languages.

Tools supporting the use of the library include

- the RGL API synopsis
- the RGL source code browser
- the RGL application expression editor

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
- PGF interpreted in C++, designed for iPhone applications

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

GF mobile application libraries

- an Android library based on Java runtime
- an iPhone library based on C++ runtime

The GF grammar compiler

Current version 3.5 (August 2013)

Backward compatibility: grammars that have worked in old versions should continue to work in newer ones

Bug tracking system:

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

GF is written in Haskell, but the binary distributions don't require any Haskell tools.

Development environments

Text editor + GF shell

- **GF modes** are available for Emacs, Geany, and Gedit

The GF-Eclipse plug-in

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

The cloud-based IDE

- http://cloud.grammaticalframework.org/

Grammar diagnostic tools in the GF shell

Relevant commands with various options:

- `i`, import
- `pg`, print grammar
- `ai`, abstract information
- `so`, show opers

Below some examples.

Verbosity

```
import -v FILE
```

gives detailed information on the compilation phases.

- if the compilation takes a long time, you can see where it gets stuck
- it shows the PGF target code size of each linearization rule.

Print words

```
print_grammar -words
```

shows the complete list of terminal tokens in the current PGF grammar.

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

counts them.

Print BNF approximation and finite automaton

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

counts the number of BNF rules

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

builds a finite automaton approximating the grammar, in graphviz.

Print missing linearizations

```
print_grammar -missing
```

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

Resource grammar tools

The structure of the resource grammar

The main API modules

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

Additional API modules

- `Symbolic`, functions for mixing text with formulas,
- `Irreg`, a list of irregular words (mostly verbs) for some languages,

- `Dict`, a large-scale morphological dictionary for some languages

- `Try`, a combination of `Syntax`, `Paradigms`, and `Lexicon`, useful for testing RGL function combinations in the GF shell, or in strictly monolingual code.

A best practice

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.

Writing a grammar

Steps

1. Create test corpus

2. Write abstract syntax

3. Write concrete syntax for one language

   - choose linearization types

   - write linearization rules

   - test, test, test!

4. Port concrete syntax to another language

5. Consider a functor

Mapping to resource grammar categories

The most useful linearization types for application grammar categories

`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

Linearization types should work in all languages

Thus avoid **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.

Porting a grammar to a new language: first steps

Assume we are porting English (Eng) to German (Ger). Then do as follows:

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:

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

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

Porting a grammar: next step

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

The 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:

Porting a grammar: concluding steps

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

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.

Sometimes one also needs:

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

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

Using a functor

Advantages:

- 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.

Disadvantages:

- 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.

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.*

To monitor your progress:

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