MOLTO

Published on *Multilingual Online Translation* (http://www.molto-project.eu)

D10.3 MOLTO web service, final version

Contract No.:	FP7-ICT-247914
Project full title:	MOLTO - Multilingual Online Translation
Deliverable:	D10.3 MOLTO web service, final version
Security (distribution level):	Public
Contractual date of delivery:	M39
Actual date of delivery:	May 2013
Type:	Prototype
Status & version:	Draft
Author(s):	Thomas Hallgren, Olga Caprotti et al.
Task responsible:	UGOT
Other contributors:	UPC, UHEL, Ontotext

Abstract

In this deliverable we document the web services that have been provided by the MOLTO project. Many of them have been released with dedicated deliverables, for those we do not enter into the specific details. Instead we focus on the web services powering some of the MOLTO flagships at the end of the project's lifetime.

1. The GF Cloud Service API

The GF Cloud Service API exposes any PGF compiled grammar as a web service via the **PGF web service API**, it provides additional functionality of some commands in the **GF shell** and some services for grammar compilation and persistent storage of files in the cloud. These features are used for instance in the implementation of the GF Simple Editor (http://cloud.grammaticalframework.org/gfse/) developed as a translators' tool during MOLTO.

1.1 Availability and Protocol

The service is available from `http://cloud.grammaticalframework.org/`. The source code for hosting a local version of the webservice is distributed in the GF distribution, hence users that have GF installed on their own computer can also run the service locally by starting GF with the parameter

```
--server[=port] Run in HTTP server mode on given port (default 41296).
```

Requests are made via HTTP with the GET or POST method. (The examples below show GET requests, but POST is preferred for requests that change the state on the server.) Data in requests is in the `application/x-www-form-urlencoded` format (the format used by default by web browsers when submitting form data). Data in responses is usually in JSON format. The HTTP response code is usually 200, but can also be 204 (after file upload), 404 (file to download or remove was not found), 400 (for unrecognized commands or missing/unacceptable parameters in requests) or 501 (for unsupported HTTP request methods). Unrecognized parameters in requests are silently ignored.

More details on how to run the service are given in Deliverable 2.3 on page 7 under "Building a web application".

1.2 PGF Service Requests

The GF Cloud Service supports a set of PGF service requests, for example, a request like

```
    http://cloud.grammaticalframework.org/grammars/Foods.pgf?command=random
```

might return a result like

```
    [{"tree":"Pred (That Pizza) (Very Boring)"}]
```

The PGF Service in the GF Cloud is the application which exposes the PGF API as Web Service. The application uses FastCGI as communication protocol to talk with the web server. The data protocol that we use is JSON. Information for how to compile and install the service could be found here.

A compiled GF grammars could be used in web applications in the same way as JSP, ASP or PHP pages are used. The compiled PGF file is just placed somewhere in the web site directory. When there is a request for access to a `.pgf` file then the web server redirects the request to the GF web service. The service knows how to load the grammar and interpret the parameters given in the URL.

If `my_grammar.pgf` is a grammar placed in the root folder of the web site for localhost then the grammar could be accessed using this URL:

```
http://localhost/my_grammar.pgf
```

Since there aren't any parameters passed in this case, the web service will respond with some general information about the grammar, encoded in JSON format. To perform specific command you have to tell what command you want to perform. The command is encoded in the parameter `command` i.e.:

```
http://localhost/my_grammar.pgf?command=cmd
```

where `cmd` is the name of the command. Usually every command also requires specific list of other arguments which are encoded as parameters as well. The list of all supported commands follows:

Commands

Grammar

This command provides some general information about the grammar. This command is also executed if no `command` parameter is given.

Input

Parameter	Description	Default
command	`grammar`	-

Output

Object with three fields:

Field	Description
name	the name of the abstract syntax in the grammar
userLanguage	the concrete language in the grammar which best matches the default language, set in the user's browser
categories	list of all abstract syntax categories defined in the grammar
functions	list of all abstract syntax functions defined in the grammar
languages	list of concrete languages available in the grammar

Every language is described with object having this two fields:

Field	Description
name	the name of the concrete syntax for the language
languageCode	the two character language code according to the ISO standard i.e. en for English, bg for Bulgarian, etc.

The language codes should be specified in the grammar because they are used to identify the user language. The web service receives the code of the language set in the browser and compares it with the codes defined in the grammar. If there is a match then the service returns the corresponding concrete syntax name. If no match is found then the first language in alphabetical order is returned.

Parsing

This command parses a string and returns a list of abstract syntax trees.

Input

Parameter	Description	Default

command	`parse`	-
cat	the start category for the parser	the default start category for the grammar
input	the string to be parsed	empty string
from	the name of the concrete syntax to use for parsing	all languages in the grammar will be tried
limit	limit how many trees are returned (gf>3.3.3)	no limit is applied

Output

List of objects where every object represents the analyzes for every input language. The objects have three fields:

Field	Description
from	the concrete language used in the parsing
brackets	the bracketed string from the parser
trees	list of abstract syntax trees
typeErrors	list of errors from the type checker

The abstract syntax trees are sent as plain strings. The type errors are objects with two fields:

Field	Description
fid	forest id which points to a bracket in the bracketed string where the error occurs
msg	the text message for the error

The current implementation either returns a list of abstract syntax trees or a list of type errors. By checking whether the field trees is not null we check whether the type checking was successful.

Linearization

The command takes an abstract syntax tree and produces string in the specified language(s).

Input

Parameter	Description	Default
command	`linearize`	-
tree	the abstract syntax tree to linearize	-
to	the name of the concrete syntax to use in the linearization	linearizations for all languages in the grammar will be generated

Output

Field	Description
to	the concrete language used for the linearization
tree	the output text

Translation

The translation is a two step process. First the input sentence is parsed with the source language and after that the output sentence(s) are produced via linearization with the target language(s). For that reason the input and the output for this command is the union of the input/output of the commands for parsing and the one for linearization.

Input

Parameter	Description	Default
command	`translate`	-
cat	the start category for the parser	the default start category for the grammar

input	the input string to be translated	empty string
from	the source language	all languages in the grammar will be tried
to	the target language	linearizations for all languages in the grammar will be generated
limit	limit how many parse trees are used (gf>3.3.3)	no limit is applied

Output

The output is a list of objects with these fields:

Field	Description
from	the concrete language used in the parsing
brackets	the bracketed string from the parser
translations	list of translations
typeErrors	list of errors from the type checker

Every translation is an object with two fields:

tree	abstract syntax tree
linearizations	list of linearizations

Every linearization is an object with two fields:

Field	Description
to	the concrete language used in the linearization
text	the sentence produced

The type errors are objects with two fields:

Field	Description
fid	forest id which points to a bracket in the bracketed string where the error occurs
msg	the text message for the error

The current implementation either returns a list of translations or a list of type errors. By checking whether the field translations is not null we check whether the type checking was successful.

Random_Generation

This command generates random abstract syntax tree where the top-level function will be of the specified category. The categories for the sub-trees will be determined by the type signatures of the parent function.

Input

Parameter	Description	Default
command	should be `random`	-
cat	the start category for the generator	the default start category for the grammar
limit	maximal number of trees generated	1

Output

The output is a list of objects with only one field:

Field	Description
tree	the generated abstract syntax tree

The length of the list is limited by the limit parameter.

Word_Completion

Word completion is a special case of parsing. If there is an incomplete sentence then it is first parsed and after that the state of the parse chart is used to predict the set of words that could follow in a grammatically correct sentence.

Input

Parameter	Description	Default
command	`complete`	-
cat	the start category for the parser	the default start category for the grammar
input	the string to the left of the cursor that is already typed	empty string
from	the name of the concrete syntax to use for parsing	all languages in the grammar will be tried
limit	maximal number of trees generated	all words will be returned

Output

The output is a list of objects with two fields which describe the completions.

Field	Description
from	the concrete syntax for this word
text	the word itself

Abstract Syntax Tree Visualization

This command renders an abstract syntax tree into image in PNG format.

Input

Parameter	Description	Default
command	`abstrtree`	-
tree	the abstract syntax tree to render	-
format	output format (gf>3.3.3)	PNG

Output

Byy default, the output is an image in PNG format. The content-type is set to `image/png`, so the easiest way to visualize the generated image is to add HTML element <img> which points to URL for the visualization command i.e.:

```
<img src="http://localhost/my_grammar.pgf?command=abstrtree&tree=..."/>
```

The output can also be in GIF ('image/gif'), SVG ('image/svg+xml') or GV (graphviz) format by setting the 'format' option

Parse Tree Visualization

This command renders the parse tree that corresponds to a specific abstract syntax tree. The generated image is in PNG format.

Input

Parameter	Description	Default
command	`parsetree`	-
tree	the abstract syntax tree to render	-
from	the name of the concrete syntax to use in the rendering	-
format	output format (gf>3.3.3)	png
options	additional rendering options (gf>3.4)	-

The additioal rendering options are: noleaves, nofun and nocat (booleans, false by default); nodefont, leaffont,nodecolor, leafcolor, nodeedgestyle and leafedgestyle (strings, have builtin defaults).

Output

By default, the output is an image in PNG format. The content-type is set to 'image/png', so the easiest way to visualize the generated image is to add HTML element <img> which points to URL for the visualization command i.e.:

```
<img src="http://localhost/my_grammar.pgf?command=parsetree&tree=..."/>
```

The output can also be in GIF ('image/gif'), SVG ('image/svg+xml') or gv (graphviz) format by setting the `format` option

---

Word Alignment Diagram

This command renders the word alignment diagram for some sentence and all languages in the grammar. The sentence is generated from a given abstract syntax tree.

Input

Parameter	Description	Default
command	`alignment`	-
tree	the abstract syntax tree to render	-
format	output format (gf>3.3.3)	PNG
to	list of languages to include in the diagram (gf>3.4)	all languages supported by the grammar

Output

By default, the output is an image in PNG format. The content-type is set to 'image/png', so the easiest way to visualize the generated image is to add HTML element `` which points to URL for the visualization command i.e.:

```
<img src="http://localhost/my_grammar.pgf?command=alignment&tree=..."/>
```

The output can also be in GIF ('image/gif'), SVG ('image/svg+xml') or GV (graphviz) format by setting the 'format' option

1.3 GF Shell Service

This service lets you execute arbitrary GF shell commands. Before you can do this, you need to use the `/new` command to obtain a working directory (which also serves as a session identifier) on the server, see below.

```
/gfshell?dir=...&command=i+Foods.pgf

/gfshell?dir=...&command=gr
     Pred (That Pizza) (Very Boring)
/gfshell?dir=...&command=ps+-lextext+%22That+pizza+is+very+boring.%22
     that pizza is very boring .
```

For documentation of GF shell commands, see:

- GF Shell Reference

Additional cloud service

`/new`
: This generates a new working directory on the server, e.g. `/tmp/gfse.123456`. Most of the cloud service commands require that a working directory is specified in the `dir` parameter. The working directory is persistent, so clients are expected to remember and reuse it. Access to previously uploaded files requires that the same working directory is used.

`/parse?`path=source
: This command can be used to check GF source code for syntax errors. It also converts GF source code to the JSON representation used in GFSE (the cloud-based GF grammar editor).

`/cloud?dir=...&command=`**`upload`**`&`path1=source1`&`path2=source2`&`...
: Upload files to be stored in the cloud. The response code is 204 if the upload was successful.

`/cloud?dir=...&command=`**`make`**`&`path1=source1`&`path2=source2`&`...
: Upload grammar files and compile them into a PGF file. Example response: `{ "errorcode":"OK", // "OK" or "Error"`

```
  "command":"gf -s -make FoodsEng.gf FoodsSwe.gf FoodsChi.gf",
  "output":"\n\n" // Warnings and errors from GF
```

```
        }
```
`/cloud?dir=...&command=`**`remake`**`&`path1=source1`&`path2=source2`&`...
: Like `command=`**`make`**, except you can leave the sourcei parts empty to reuse previously uploaded files.

`/cloud?dir=...&command=`**`download`**`&file=`path
: Download the specified file.

`/cloud?dir=...&command=`**`ls`**`&ext=.pgf`
: List files with the specified extension, e.g. `["Foods.pgf","Letter.pgf"]`.

`/cloud?dir=...&command=`**`rm`**`&file=`path
: Remove the specified file.

`/cloud?dir=...&command=`**`link_directories`**`&newdir=...`
: Combine server directores. This is used by GFSE to share grammars between multiple devices.

1.4 Examples

GF can be used interactively from the GF Shell. Some of the functionality availiable in the GF shell is also available via the GF web services API.

The GF Web Service API page describes the calls supported by the GF web service API. Below, we illustrate these calls by examples, and also show how to make these calls from JavaScript using the API defined in `<a href="js/pgf_online.js" rel="nofollow">pgf_online.js</a>`.

Note that `pgf_online.js` was initially developed with one particular web application in mind (the minibar), so the server API was incomplete. It was simplified and generalized in August 2011 to support the full API.

These boxes show what the calls look like in the JavaScript API defined in `pgf_online.js`. These boxes show the corresponding URLs sent to the PGF server. These boxes show the JSON (JavaScript data structures) returned by the PGF server. This will be passed to the callback function supplied in the call.

Initialization

// Select which server and grammars to use:

```
var server_options = {
                             grammars_url: "http://www.grammaticalframework.org/grammars/",
                             grammar_list: ["Foods.pgf"] // It's ok to skip this
                             }
var server = pgf_online(server_options);
```

Examples

// Get the list of available grammars

```
server.get_grammarlist(callback)
http://localhost:41296/grammars/grammars.cgi
["Foods.pgf","Phrasebook.pgf"]
```

// Select which grammar to use

```
server.switch_grammar("Foods.pgf")
```

// Get list of concrete languages and other grammar info

```
server.grammar_info(callback)
http://localhost:41296/grammars/Foods.pgf
       {"name":"Foods",
        "userLanguage":"FoodsEng",
        "startcat":"Comment",
        "categories":["Comment","Float","Int","Item","Kind","Quality","String"],
        "functions":["Boring","Cheese","Delicious","Expensive","Fish","Fresh",
                     "Italian","Mod","Pizza","Pred","That","These","This","Those","Very",
                     "Warm","Wine"],
        "languages":[{"name":"FoodsBul","languageCode":""},
                     {"name":"FoodsEng","languageCode":"en-US"},
                     {"name":"FoodsFin","languageCode":""},
                     {"name":"FoodsSwe","languageCode":"sv-SE"},
                     ...]
          }
```

// Get a random syntax tree

```
server.get_random({},callback)
http://localhost:41296/grammars/Foods.pgf?command=random
       [{"tree":"Pred (That Pizza) (Very Boring)"}]
```

// Linearize a syntax tree

```
server.linearize({tree:"Pred (That Pizza) (Very Boring)",to:"FoodsEng"},callback)
http://localhost:41296/grammars/Foods.pgf?command=linearize&amp;tree=Pred+(That+Pizza)+(Very+Boring)&amp;to=FoodsEng
       [{"to":"FoodsEng","text":"that pizza is very boring"}]
       server.linearize({tree:"Pred (That Pizza) (Very Boring)"},callback)

 http://localhost:41296/grammars/Foods.pgf?command=linearize&amp;tree=Pred+(That+Pizza)+(Very+Boring)
     [{"to":"FoodsBul","text":"онази пица е много еднообразна"},
     {"to":"FoodsEng","text":"that pizza is very boring"},
     {"to":"FoodsFin","text":"tuo pizza on erittäin tylsä"},
     {"to":"FoodsSwe","text":"den där pizzan är mycket tråkig"},
     ...
     ]
```

// Parse a string

```
server.parse({from:"FoodsEng",input:"that pizza is very boring"},callback)
http://localhost:41296/grammars/Foods.pgf?command=parse&amp;input=that+p...
       [{"from":"FoodsEng",
          "brackets":{"cat":"Comment","fid":10,"index":0,
         "children":[{"cat":"Item","fid":7,"index":0,
         "children":[{"token":"that"},{"cat":"Kind","fid":6,"index":0,
         "children":[{"token":"pizza"}]}]},
        {"token":"is"},{"cat":"Quality","fid":9,"index":0,
          "children":[{"token":"very"},{"cat":"Quality","fid":8,"index":0,
          "children":[{"token":"boring"}]}]}]},
          "trees":["Pred (That Pizza) (Very Boring)"]}]
```

// Translate to all available languages

```
server.translate({from:"FoodsEng",input:"that pizza is very boring"},callback)
...
```

// Translate to one language

```
server.translate({input:"that pizza is very boring", from:"FoodsEng", to:"FoodsSwe"}, callback)
http://localhost:41296/grammars/Foods.pgf?command=translate&amp;input=th...
      [{"from":"FoodsEng",
        "brackets":{"cat":"Comment","fid":10,"index":0,
        "children":[{"cat":"Item","fid":7,"index":0,
        "children":[{"token":"that"},{"cat":"Kind","fid":6,"index":0,
        "children":  [{"token":"pizza"}]}]},{"token":"is"},{"cat":"Quality","fid":9,"index":0,
        "children":[{"token":"very"},{"cat":"Quality","fid":8,"index":0,"children":[{"token":"boring"}]}]}]},
        "translations":
        [{"tree":"Pred (That Pizza) (Very Boring)",
          "linearizations":
           [{"to":"FoodsSwe",
              "text":"den där pizzan är mycket tråkig"}]}]}]
```

// Get completions (what words could come next)

```
server.complete({from:"FoodsEng",input:"that pizza is very "},callback)
http://localhost:41296/grammars/Foods.pgf?command=complete&amp;input=tha...
      [{"from":"FoodsEng", "brackets":{"cat":"_","fid":0,"index":0,
        "children":[{"cat":"Item","fid":7,"index":0,
        "children":[{"token":"that"},{"cat":"Kind","fid":6,"index":0,
        "children":[{"token":"pizza"}]}]},{"token":"is"},{"token":"very"}]},
        "completions":["boring","delicious","expensive","fresh","Italian","very","warm"],
        "text":""}]
```

// Get info about a category in the abstract syntax

```
server.browse({id:"Kind"},callback)
http://localhost:41296/grammars/Foods.pgf?command=browse&amp;id=Kind&amp...
      {"def":"cat Kind", "producers":["Cheese","Fish","Mod","Pizza","Wine"],
       "consumers":["Mod","That","These","This","Those"]}
```

// Get info about a function in the abstract syntax

```
server.browse({id:"This"},callback)
http://localhost:41296/grammars/Foods.pgf?command=browse&amp;id=This&amp...
       {"def":"fun This : Kind -&gt; Item","producers":[],"consumers":[]}
```

// Get info about all categories and functions in the abstract syntax

```
server.browse({},callback)
http://localhost:41296/grammars/Foods.pgf?command=browse&amp;format=json
       {"cats":{"Kind":{"def":"cat Kind",
             "producers":["Cheese","Fish","Mod","Pizza","Wine"],
             "consumers":["Mod","That","These","This","Those"]},
     ...},
        "funs":{"This":{"def":"fun This : Kind -&gt; Item","producers":[],"consumers":[]},
     ...}
     }
```

// Convert an abstract syntax tree to JSON

```
server.pgf_call("abstrjson",{tree:"Pred (That Pizza) (Very Boring)"},callback)
```

```
http://localhost:41296/grammars/Foods.pgf?command=abstrjson&amp;tree=Pred+(That+Pizza)+(Very+Boring)
       {"fun":"Pred","fid":4,
        "children":[{"fun":"That","fid":1,
          "children":[{"fun":"Pizza","fid":0}]},
         {"fun":"Very","fid":3,
          "children":[{"fun":"Boring","fid":2}]}]}
```

2. MOLTO Application Grammars

At the beginning of the project, we have published the MOLTO Phrasebook as example application grammar. For the final version of our online service, we show all the relevant GF application grammars that have been developed in various work-packages as supporting grammars for larger applications. Each example in this collection can be used by a new GF grammar developer as a starting point that can be further extended. In this deliverable we briefly document the grammars, the online applications that use them, and give quick hints on where extension can occur in future work.

The Geography Grammar

This grammar has been developed originally for the semantic multilingual wiki system AceWiki-GF, as documented in Deliverable D11.3. The grammar can be used online at http://attempto.ifi.uzh.ch/acewiki-gf/.

It currently supports 3 languages: ACE, German and Spanish, where ACE is a formal language used for automated reasoning. A 500-word geography domain vocabulary has been created to describe Europe.

ACE is represented by two languages, Ace and Ape. Ape linearizations contain explicit lexical entries so that the ACE parser (APE) can be used to map the sentences of this grammar to OWL. The wiki shows how this mapping works.

The source for the grammar is distributed at Github at: https://github.com/Attempto/ACE-in-GF.

The MOLTO Phrasebook

The MOLTO Phrasebook has been the first demonstrator of the features of the Grammatical Framework technology, online since M3 of the project's lifetime. The application grammar was designed to serve as model for best practices. It shows a modular approach to the definition of abstract types and functions from the domain of travelers' phrasebooks , covering natural language for giving directions, ordering a meal, and greeting friends. It has categories for Citizenship, Country, Currency, Date and week Day, Digits, DrinkKind and MassKind, Languages, Greetings and many more. Eng, Bul, Cat, Dan, Dut, Fin, Fre, Ger, Hin, Ita, Lav, Nor, Pes, Pol, Ron, Rus, Spa, Swe, Tha, Urd. It has a module that handles disambiguation in Eng and in Ron.

The final version is online at http://www.molto-project.eu/cloud/gf-application-grammars by selecting as application `Phrasebook.pgf`.

The repository for the grammar file itself is at http://www.molto-project.eu/biblio/software/phrasebookpgf.

The Mathematical Grammars

MathBar.pgf

`MathBar.pgf` is the application grammar developed for the mathematical natural language domain. It supports the following languages: Fre, Cat, Spa, Eng and Fin. More languages are available but have not been checked against quality. The Mathematical Grammar Library (MGL) is a specialized language in which textual fragments are interspersed with formal fragments represented in the typesetting language LaTeX.

The source files are distributed via svn at URL: svn://molto-project.eu/mgl Repository Root: svn://molto-project.eu Repository UUID: 54d65b75-f25a-4862-968f-dc0a3298bc6b Revision: 2432

The compiled PGF grammar is available from http://www.molto-project.eu/biblio/software/mathbarpgf.

Commands.pgf

`Commands.pgf` is the application grammar developed for natural language I/O to the Sage computer algebra system. It translates input queries and output answers into natural language of mathematical nature. Users can ask for computations related to arithmetic, domain and range of functions, differentiation and integration. It also supports the usage of referential mechanism by the pronoun `it`, which will link to the previous result in a session of sequential computations. English, German and Spanish are currently supported.

Dialog.pgf

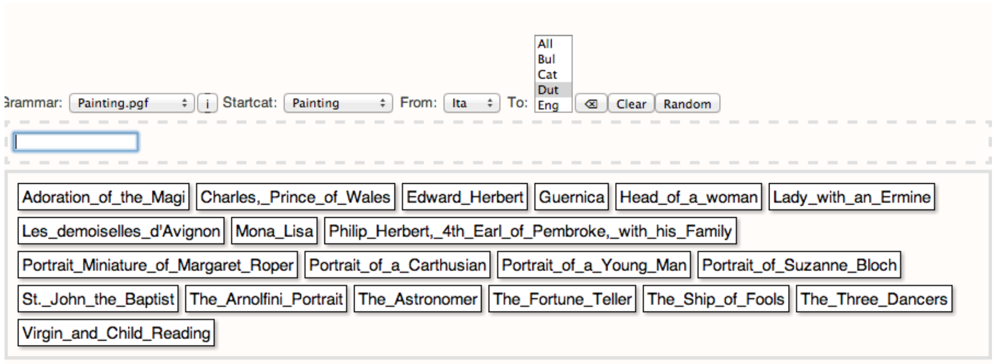
`Dialog.pgf` translates natural language interactions of the word problems prototype documented in Deliverable D6.3. It is used to give hints in the student's language and to formalize the students' answers or commands as Prolog statements that can be reasoned automatically with. It is an example of how a description of a specific world situation (owing fruits, animal in a farm) can be interpreted and formalized. Catalan, English, Spanish and Swedish are currently supported. The programming language Prolog is also supported. SVN info for compilation from source: URL: svn://molto-project.eu/mgl/wproblems Repository Root: svn://molto-project.eu Repository UUID: 54d65b75-f25a-4862-968f-dc0a3298bc6b Revision: 2432 GF version compilation: Grammatical Framework (GF) version 3.4-darcs.

The version archived and deployed on the MOLTO cloud is http://www.molto-project.eu/biblio/software/dialogpgf.

The Painting Grammar

The work-package dealing with the domain of cultural heritage has focused on the description of museum artefacts, in particular paintings. While the description of the subject matter of a painting is an open domain, the other characteristics of a painting can be described by a constrained natural language tightly coupled with the underlying knowledge representation used by museum curators. The design of this grammar has been based on sample descriptions of paintings retrieved from the Gothenburg City Museum and has been further applied to generate descriptions of artefacts stored on public web pages, such as DBPedia.

One major discussion has concerned the identification of entity names, museum names, as well as famous painters' or masterpieces are often translated ad hoc. For such cases, it is hard to create grammar-based translation rules, consider for instance **Mona Lisa**, in Italian often referred to as **La Gioconda**. The approach taken in this work package has been that of not translating the entity names found in the knowledge base while investigating whether historically there could be a given title or name that could be taken as a universally valid identifier for that entity. Since to our knowledge, there seems to be no agreement by museum curators on unique resource identifiers (whereas for instance, in the publishing world, there have been efforts of uniquely indexing published material), we have adopted a naming based on the resource descriptors we retrieved in our samples. In terms of future web application building, we are aware that resource identification and/or retrieval by the common name is not as sound as by unique ID.


Grammar: Painting.pgf Startcat: Painting From: Ita To: All Bul Cat Dut Eng

This grammar is also modularly designed and assembled categories that are used to represent location, material, color, dimension, type of work, and painter's biographical data. The most relevant feature of this grammar is the construction of a description as a sequence of phrases related to the same artefact, using referential chains to build up a coherent discourse. Please see the list of publications tagged with WP8 for further information about the comparative study of texts in the cultural heritage domain and about the background knowledge base underlying the ontology from which texts in 15 languages are generated.

The grammar files are avliable on svn: molto-project.eu/wp8/d8.3/grammars/

The demo webpage is avaliable at: http://museum.ontotext.com/

Grammar characteristics

The version of the grammar on display at the MOLTO Application Grammar web service (TextPainting.pgf) features:

- The following start categories: Main category: Description 9 semantic categories which represent the ontology classes: Colour, Material, Museum, Painter, Painting, PaintingType, Size, Title, and Year. Of these 8 categories, 5 are optional, hence the additional 'Opt' categories. 3 category types: String, Int, Float 1 grammatical category for creating nested colour strings: ListColour
- Support for 15 languages: Bulgarian (Bul), Catalan (Cat), Danish (Dan), Dutch (Dut), English (Eng), Finnish (Fin), French (Fre), Hebrew (Heb), Italian (Ita), German (Ger), Norwegian (Nor), Romanian (Rom), Russian (Rus), Spanish (Spa), Swedish (Swe).
- Up to three sentence long text generation where each sentence may be constructed with different semantic categories. For example, consider the first sentence of a description:

```
Forest[PAINTING] was painted by Paul Cezanne[PAINTER] in 1902[YEAR].

Forest[PAINTING] was painted on canvas[MATERIAL] by Paul Cezanne[PAINTER] in 1902[YEAR].
```

- Change of the syntactic element of the reference entity in sentence initial, i.e.

  `Forest was painted by Paul Cezanne in 1902. It[Pronoun] is painted in green and blue.`

  `Forest was painted by Paul Cezanne in 1902. This painting[NounPhrase] is displayed at the National Gallery of Canada.`

Restrictions of the grammar

As mentioned above, the names of the paintings and painters have been left untranslated. Since museum names have been translated automatically, some translations are missing. Therefore two or three words names contain underscores.

Hebrew texts with names that are missing translations cause wrong ordering of the words in a sentence.

The Patent Query Grammar

This grammar is used to translate user queries into SPARQL. It contains 4 languages: English, German, French and a concrete syntax corresponding to SPARQL. Since the grammar is adapted to the patents domain, the constructors from the abstract syntax describe individual queries that depend on the domain. So, the SPARQL mappings are written in a gap-filling fashion, by specifying the query with spaces for the arguments.

Mode details from deliverables released by Work-package 7.

The sources are in the svn://molto-project.eu/wp7/query/grammars.

The Words300 Grammar

The Words300-grammar was produced to evaluate the correctness of the multilingual translation of ACE sentences offered by the ACE-in-GF grammar. The grammar contains ~300 words from the GF resource grammar library (RGL), namely the words from the ACE word classes common noun, transitive verb and proper name. Currently, most of the RGL languages are included, altogether 21 languages.

Note that the English sentences that this grammar produces are not always valid ACE sentences, due to "spaces in content words" which is not allowed in ACE. For example, the grammar supports `For which computer does John wait?` while ACE requires `Which computer does John wait-for?`.

The source for the grammar is hosted at Github: https://github.com/Attempto/ACE-in-GF

The grammar can be used in a wiki at: http://attempto.ifi.uzh.ch/acewiki-gf/gf/Words300/main/

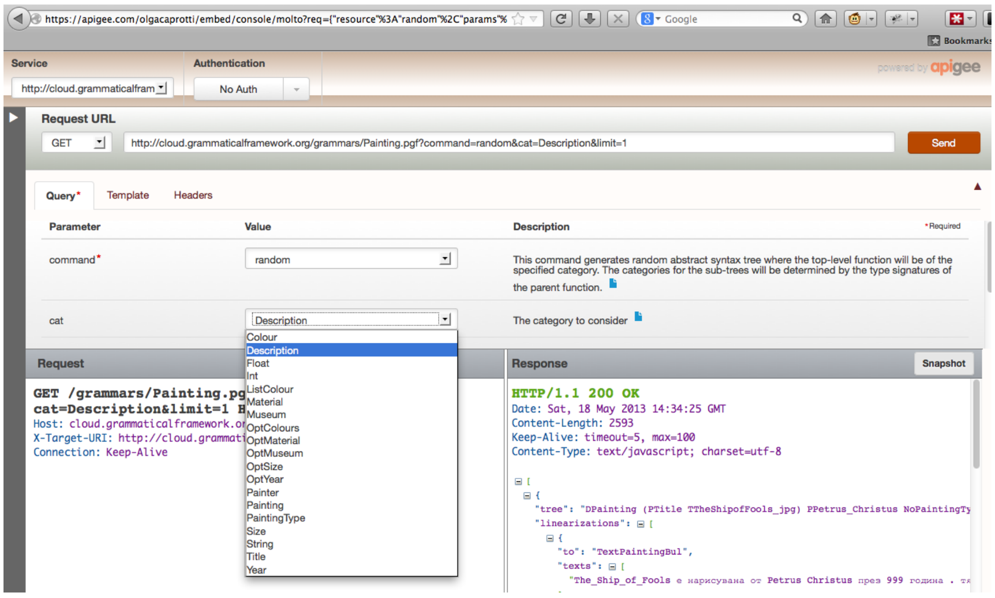
For the description of the above-mentioned evaluation, see D11.3.

3. Sample WADL for a GF Application Grammar

The Web Application Description Language, WADL (http://www.w3.org/Submission/wadl/), is a specification language of HTTP-based Web applications that can be read and processed automatically to generate web service clients. In combination with an API platform, such as Apigee (http://apigee.com), it is possible to expose the API of a web service to developers of third-party web applications so they can quickly integrate with further services, for instance authentication, logging data, performance monitoring.

For a GF grammar developer, writing a WADL specification for the grammar is a quick way to expose the translation command invocation details in a machine processable way. Any PGF compiled GF grammar can be fed to the GF Web Service along with specific commands and query parameters to provide for instance parsing, linearization, and random tree generation according to the the GF Web Service API. The documentation is available at http://code.google.com/p/grammatical-framework/wiki/GFWebServiceAPI. The web application running the GF web service is distributed in the regular GF distribution. A Java frontend was developed during the MOLTO project, http://www.molto-project.eu/biblio/software/gf-java-master, and is being maintained at Github, https://github.com/Kaljurand/GF-Java.

The example WADL specification file for web services powered by the `TextPainting.pgf` grammar hosted on the Grammatical Framework cloud server and deployed on Apigee, as seen in the figure below, is available at http://www.molto-project.eu/biblio/web-service/textpaintingpgf. It exposes the GET command for retrieving the grammar information and the GET command for retrieving a random production in any of the available categories.



Service: http://cloud.grammaticalfram

Authentication: No Auth

Request URL

GET http://cloud.grammaticalframework.org/grammars/Painting.pgf?command=random&cat=Description&limit=1

Parameter	Value	Description
command*	random	This command generates random abstract syntax tree where the top-level function will be of the specified category. The categories for the sub-trees will be determined by the type signatures of the parent function.
cat	Description	The category to consider

*Required

Request

```
GET /grammars/Painting.pg
cat=Description&limit=1 H
Host: cloud.grammaticalframework.or
X-Target-URI: http://cloud.grammati
Connection: Keep-Alive
```

Response

```
HTTP/1.1 200 OK
Date: Sat, 18 May 2013 14:34:25 GMT
Content-Length: 2593
Keep-Alive: timeout=5, max=100
Content-Type: text/javascript; charset=utf-8

⊟ [
 ⊟ {
   "tree": "DPainting (PTitle TTheShipofFools_jpg) PPetrus_Christus NoPaintingTy
   "linearizations": ⊟ [
     ⊟ {
       "to": "TextPaintingBul",
       "texts": ⊟ [
         "The_Ship_of_Fools е нарисувана от Petrus Christus през 999 година . тя
```

The designer of the web service for translating painting descriptions might decide to expose a very specific command, for instance only parsing of descriptions in Italian. This is possible by selecting what to describe in the WADL specification in a careful way, by not exposing the full generality of the grammar. Grammars that are stable only in certain categories, for instance because of increasing complexity in their modular stepwise development, can in this way be deployed while under development, provided the only web services exposed are the stable ones.

5. Future work

GF compiled grammars deployed as web services seem to be able to offer valuable translation and parsing functionality to developers of online applications. With the work done during the MOLTO project we have only begun to experiment with the usage of GF powered web services and the results have been positive.

To further the adoption of GF and MOLTO technologies for high-quality translation of web applications, it would be important to be able to obtain the machine processable specification of the services, for instance as WADL or SOAP, directly available as an export command in the GF Web Service API. The client applications for the web services exposed by the application grammar would then be generated automatically allowing very fast prototyping. Software that generates web clients based on SOAP or WADL is already existing.

Source URL: http://www.molto-project.eu/wiki/living-deliverables/d103-molto-web-service-final-version