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

D2.2 Grammar IDE

Contract No.:	FP7-ICT-247914
Project full title:	MOLTO [1] - Multilingual Online Translation
Deliverable:	D2.2 Grammar IDE
Security (distribution level):	Public
Contractual date of delivery:	M18
Actual date of delivery:	September 2011
Type:	Prototype
Status & version:	Final
Author(s):	A. Ranta, T. Hallgren, et al.
Task responsible:	UGOT [2]
Other contributors:	John Camilleri, Ramona Enache

Abstract

Deliverable D2.2 describes the functionalities for an Integrated Development Environment (IDE) for GF [3] (Grammatical Framework). The main question it addresses is 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 [3] programming in the cloud, and an Eclips plug-in, targeted for expert users working with large projects, which may involve the integration of GF [3] with other components. Example-based grammar writing is also described in the end.

1. Introduction

An IDE, Integrated Development Environment, is a software system targeted for programmers. It helps the programmer to write code and test and maintain it. The tasks where IDE helps can include

- editing source code
- compiling and/or interpreting the code
- maintain complex projects with their module dependencies
- test and debug the code

(cf. http://en.wikipedia.org/wiki/Integrated_development_environment [4]). While interactive command interpreters (such as Unix shells) were historically the first systems recognized as IDE's, the contemporary notion of "IDE's proper" assumes a set of visual tools. The most widely used IDE's are probably Eclipse (http://www.eclipse.org/ [5]), Microsoft Visual Studio (http://www.microsoft.com/visualstudio [6]), and Apple's XCode (http://developer.apple.com/xcode/ [7]). Each of these is a desktop program of substantial size. But recent times have also seen Web IDE's (WIDE), where the user can write programs without implementing any software locally; an example is CodeRun (http://www.coderun.com [8]).

The purpose of this document is to introduce an IDE for GF [3], Grammatical Framework (http://www.grammaticalframework.org/ [9]). GF [3] is a programming language designed for writing multilingual grammars and their applications (Ranta 2011). Typical applications are translation systems (with many simultaneous languages) and the localization of natural language processing systems such as question answering (with many alternative languages).

This paper will introduce two IDE's for GF [3]:

- a Web IDE (http://www.grammaticalframework.org/demos/gfse/ [10]), which allows users to build GF [3] grammars and run them "in the cloud"
- an Eclipse plug-in, which allows users to build GF [3] grammars on their desktop and link them with other Eclipse-enabled software, such as Android mobile applications and ontology engineering tools.

The Web IDE is intended to be a quick way to use GF [3], since it doesn't require any software installation, and also has some helpful functionalities to guide novice users. But it is less adapted for large GF [3] programs consisting of large numbers of modules, such as GF [3] grammar libraries. The Web IDE is a mature program tested by many users, but new developments are still expected.

The Eclipse plug-in is meant for power users of GF [3], who have to maintain perhaps hundreds of GF [3] modules simultaneously and to link them with other software. But it is less quick to get started with, since it requires the installation of both the GF [3] compiler, the Eclipse platform, and the GF [3] Eclipse plug-in. The Eclipse plug-in is still in the beginning of its development.

Both these tools are new, and have been built during 2011 within the MOLTO [1] project. The traditional "IDE" for GF [3] is one familiar from the Unix environment:

- a batch compiler
- text editor support (e.g. syntax highlighting in Emacs, Gedit, and Geany, http://www.grammaticalframework.org/doc/gf-editor-modes.html [11])
- an interactive command-based shell, "the original GF [3] program".

The interactive shell is a Read-Eval-Print loop similar to LISP and, more recently, Haskell (GHCI). While it has more IDE functionalities than many programming languages provide, we are not calling it an IDE, but reserve that name to the graphical Web IDE and Eclipse systems. Actually, the GF [3] shell can be seen as an API (Application Programmer's Interface) to the GF [3] compiler. It provides a set of commands that can be used for compiling, diagnosing, and testing GF [3] grammars. More sophisticated IDE's can be built by using the shell command language to communicate with the compiler. The document The GF [3] Grammar Compiler API (MOLTO [1] Deliverable 2.1) gives more information on the available functionalities.

2. Multilingual grammars

A GF [3] program is a multilingual grammar, which for

n languages consist of 1+n modules: one abstract syntax defining the semantic content in a language-independent way, and for each language a concrete syntax showing how this content is expressed in that language. Here is a "hello world" example for English, Finnish, and Italian:

```
abstract Hello = {
  cat Greeting ; Recipient ;
  fun
    Hello : Recipient -> Greeting ;
    World, Mum, Friends : Recipient ;
}

concrete HelloEng of Hello = {
  lin
    Hello rec = "hello" ++ rec ;
    World = "world" ;
    Mum = "mum" ;
    Friends = "friends" ;
}

concrete HelloFin of Hello = {
  lin
    Hello rec = "terve" ++ rec ;
    World = "maailma" ;
    Mum = "äiti" ;
    Friends = "ystävät" ;
}

concrete HelloIta of Hello = {
  lin
    Hello rec = "ciao" ++ rec ;
    World = "mondo" ;
    Mum = "mamma" ;
    Friends = "amici" ;
}
```

The GF [3] compiler produces from this code a system that can parse phrases like hello world, ciao mamma and also generate them each language, thus enabling translation between any pair of languages.

The Hello grammar is of course extremely simple, on purpose. But it shows the essential structure of multilingual grammars, and it is easy to see how the grammar could be extended by adding new functions (i.e. combination rules like `Hello` and words like `Mum`).

The GF [3] compiler controls that the abstract and concrete syntaxes are in synchrony. For instance, it checks that each abstract syntax function (`fun`) actually has a linearization (`lin`) in each concrete syntax. An IDE is expected to go one step further: it reminds the programmer, prior to running the compiler, of those linearizations that are missing. And when a new language (i.e. a new concrete syntax) is added to the system, the IDE initializes its code with a template for all required linearization rules.

Multilinguality is one aspect of GF [3]'s module system: each language, as well as the abstract syntax, has its own module. Larger GF [3] applications have an additional complexity created by the inheritance and opening of modules; a large grammar can easily have 20 modules involved for each language, and this is multiplied by the number of languages plus one for the abstract syntax. While the opening and inheritance correspond to the module dependencies found in most other programming languages (such as inheritance and the use of libraries), the multilinguality aspect is an extra dimension, which makes GF [3] programs more complex than usual programs.

A GF [3] project with 15 languages, as targeted in the MOLTO [1] project, involves hundreds of modules in scope at

the same time. These are roughly divided to two groups,

- the application grammar: the code written by the programmer
- the resource grammar: the code imported from libraries

The total resource grammar code in September 2011 comprises 755 modules, addressing 20 natural languages. This code is normally distributed in binaries (although the source is also available) and never read or written by the application programmer. But the programme of course needs to inspect the code: to see, for instance, what functions are available to contruct objects of a given type such as noun or sentence. Inspecting the library code is one of the most important things that should be supported by the IDE.

3. The Web IDE

Traditionally, GF [9] grammars are created in a text editor and tested in the GF [3] shell. Text editors know very little (if anything) about the syntax of GF [3] grammars, and thus provide little guidance for novice GF [3] users. Also, the grammar author has to download and install the GF [3] software on his/her own computer.

In contrast, the GF online editor for simple multilingual grammars [10] is available online, making it easier to get started. All that is needed is a reasonably modern web browser. Even Android and iOS devices can be used.

The editor also guides the grammar author by showing a skeleton grammar file and hinting how the parts should be filled in. When a new part is added to the grammar, it is immediately checked for errors.

Editing operations are accessed by clicking on editing symbols embedded in the grammar display: + = Add an item, × = Delete an item, % =Edit an item. These are revealed when hovering over items. On touch devices, hovering is in some cases simulated by tapping, but there is also a button at the bottom of the display to "Enable editing on touch devices" that reveals all editing symbols.

In spite of its name, the editor runs entirely in the web browser, so once you have opened the web page, you can continue editing grammars even while you are offline.

3.1 Status

At the moment, the editor supports only a small subset of the GF [3] grammar notation. Proper error checking is done for abstract syntax, but not (yet) for concrete syntax.

The grammars created with this editor always consists of one file for the abstract syntax, and one file for each concrete syntax.

3.1.1. ABSTRACT SYNTAX

The supported abstract syntax corresponds to context-free grammars (no dependent types). The definition of an abstract syntax is limited to

- a list of category names, Cat_1 ; ... ; Cat_n,
- a list of functions of the form Fun : Cat_1 -> ... -> Cat_n
- and a start category.

Available editing operations:

- Categories can be added, removed and renamed. When renaming a category, occurences of it in function types will be updated accordingly.
- Functions can be added, removed and edited. Concrete syntaxes are updated to reflect changes.
- Functions can be reordered using drag-and-drop.

Error checks:

- Syntactically incorrect function definitions are refused.
- Semantic problem such as duplicated definitions or references to undefined categories, are highlighted.

3.1.2. CONCRETE SYNTAX

At the moment, the concrete syntax for a language L is limited to

- opening the Resource Grammar Library modules `SyntaxL` and `ParadigmsL`, `LexiconL` and `ExtraL`,
- linearization types for the categories in the abstract syntax,
- linearizations for the functions in the abstract syntax,
- parameter type definitions, P = C_1 | ... |C_n,
- and operation definitions, op = expr, op : type = expr,

Available editing operations:

- The LHSs of the linearization types and linearizations are determined by the abstract syntax and do not need to be entered manually. The RHSs can be edited.
- Parameter types can be added, removed and edited.
- Operation definitons can be added, removed and edited.
- Definitions can be reordered (using drag-and-drop)

Also,

- When a new concrete syntax is added to the grammar, a copy of the currently open concrete syntax is created, since copying and modifying is usually easier than creating something new from scratch. (If the abstract syntax is currently open, the new conrete syntax will start out empty.)
- When adding a new concrete syntax, you normally pick one of the supported languages from a list. The language code and the file name is determined automatically. But you can also pick Other from the list and change the language code afterwards to add a concrete syntax for a language that is not in the list.

Error checks:

- The RHSs in the concrete syntax are checked for syntactic correctness by the editor as they are entered. (TODO: the syntax of parameter types is not check at the moment.)
- Duplicated definitions are highlighted. Checks for other semantic errors are delayed until the grammar is compiled.

3.2. Compiling and testing grammars

When pressing the Compile button, the grammar will be compiled with GF [3], and any errors not detected by the editor will be reported. If the grammar is free from errors the user can then test the grammar by clicking on links to the online GF [3] shell, the Minibar or the Translation Quiz.

3.3. Grammars in the cloud

While the editor normally stores grammars locally in the browser, it is also possible to store grammars in the cloud. Grammars can be stored in the cloud just for backup, or to make them accessible from multiple devices.

There is no automatic synchronization between local grammars and the cloud. Instead, the user should press  to upload the grammars to the cloud, and press  to download grammars from the cloud. In both cases, complete grammars are copied and older versions at the destination will be overwritten. When a grammar is deleted, both the local copy and the copy in the cloud is deleted.

Each device is initially assigned to its own unique cloud. Each device can thus have its own set of grammars that are not available on other devices. It is also possible to merge clouds and share a common set of grammars between multiple devices: when uploading grammars to the cloud, a link to this grammar cloud appears. Accessing this link from another device will cause the clouds of the two devices to be merged. After this, grammars uploaded from one of the devices can be downloaded on the other devices. Any number devices can join the same grammar cloud in this way.

Note that while it is possible to copy grammars between multiple devices, there is no way to merge concurrent edits from multiple devices. If the same grammar is uploaded to the cloud from multiple devices, the last upload wins. Thus the current implementation is suitable for a single user switching between different devices, but not recommended for sharing grammars between multiple users.

Also note that each grammar is assigned a unique identity when it is first created. Renaming a grammar does not change its identity. This means that name changes are propagated between devices like other changes.

3.4. Future work

This prototype gives an idea of how a web based GF [3] grammar editor could work. While this editor is implemented in JavaScript and runs in the web browser, we do not expect to create a full implementation of GF [3] that runs in the web browser, but let the editor communicate with a server running GF [3].

By developing a GF [3] server with an appropriate API, it should be possible to extend the editor to support a larger fragment of GF [3], to do proper error checking and make more of the existing GF [3] shell functionality accessible directly from the editor.

The current grammar cloud service is very primitive. In particular, it is not suitable for multiple users developing a grammar in collaboration.

3.5. Related documents

- The GF Grammar Compiler API [12].
- Slides from a presentation at the MOLTO meeting in Göteborg, March 2011 [13].

4. The Eclipse plug-in

The aim behind developing a desktop IDE for GF [3] is to provide more powerful tools than may be possible and/or practical in a web-based setting. In particular, the ability to resolve cross-references between source files and libraries instantaneously during development time is one of the primary goals and motivations for the project.

The choice was made to develop this desktop IDE as a plugin for the Eclipse Platform as it seemed to be the most popular choice among the GF [3] developer community. Support for the platform is vast and many tools for adapting Eclipse to domain-specific languages already exist. Unlike the zero-click WIDE approach, using the GF [3] Eclipse plugin (GFEP) will require some manual installation and configuration on the development machine. Thus the GFEP is aimed more at seasoned developers rather than just the curious.

4.1. Features

Implemented (including partially)

1. Syntax highlighting and error detection
2. Code folding, quick block-commenting, automatic code formatting
3. Definition outlining, jump to declaration, find usage
4. Warnings for problems in module dependancy hierarchy
5. Launch configurations, i.e. compilation directly from IDE

Coming soon

1. Auto-completion for declared identifiers
2. Inline documentation for function calls, overloads
3. Quick-fix suggestions for syntax and naming errors
4. Code generation for concrete/instance modules
5. Code generation for new languages in application grammars
6. Grouping of concrete syntaxes by language, fast switching and linked navigation
7. Built-in library browser (in particular for GF [3] resource grammar library)

Long-term goals

1. Test-suite functionality
    - Treebank management and testing
    - Possibility to incorporate treebank tool demonstrated by Jordi Saludes in the Math Grammar Library
2. Provide a single platform for developing and using embedded grammars
3. Integration with ontology engineering tools

4.2. Status

The starting point for the GFEP is using the Xtext DSL Framework for Eclipse (http://www.eclipse.org/Xtext/ [14]). By converting the GF [3] grammar into the appropriate Extended-BNF form required by the LL(*) ANTLR parser, the framework provides a good starting point for future plugin development, already including a variery of syntax checking tools and some cross-reference resolution support. The specific requirements of the GF [3] language, particularly in the way of its

special module hierarchy, mean that significant customisations to this generated base plugin are needed.

As of 1st October 2011, a first prototype of the GFEP has been released to GF [3] developers to gather some initial feedback. This first release is not intended to be a mature development tool, but a showcase of some of the potential features that can be provided by developing GF [3] grammars within a powerful desktop IDE. Reactions from within the GF [3] developer community will guide the way forward, both in prioritizing the future tasks and also in better guaging the person-month cost that an eventual mature version of the plugin would require.

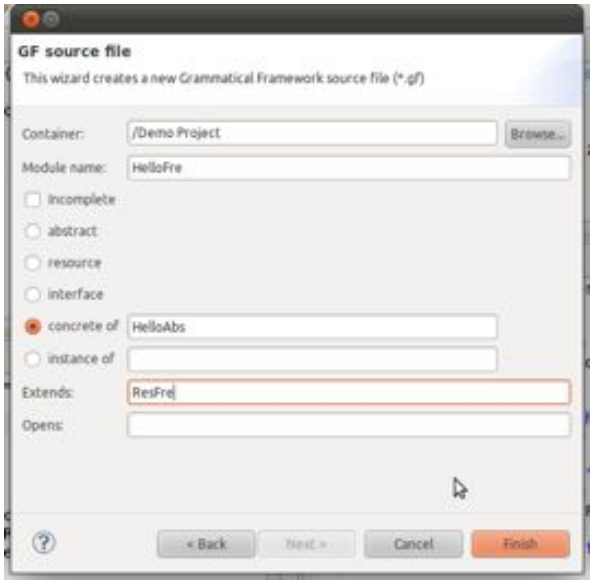
4.3. Trying out the GFEP prototype

INSTALLATION

1. Eclipse is of course required. The plugin was developed using Eclipse 3.7 but older versions should also work.
2. Inside Eclipse, go to Help > Install New Software.
3. Add new software site using the URL: http://www.grammaticalframework.org/eclipse/beta/ [15].
4. Select the GF [3] Eclipse Plugin, click Next, accept the license agreement and install. If it takes a long time to calculate dependencies, just be patient. I'm not yet sure if this is an abnormal issue or not.
5. Accept the prompt warning that the software is unsigned.
6. Restart Eclipse when prompted.
7. (Optional) Add the GF [3] perspective clicking Open Perspective > Other.
8. (Optional) Go to Run > Run Configurations and add a new Grammatical Framework configuration. Fill in the necessary fields in the Main tab, and click Apply to save the new configuration.

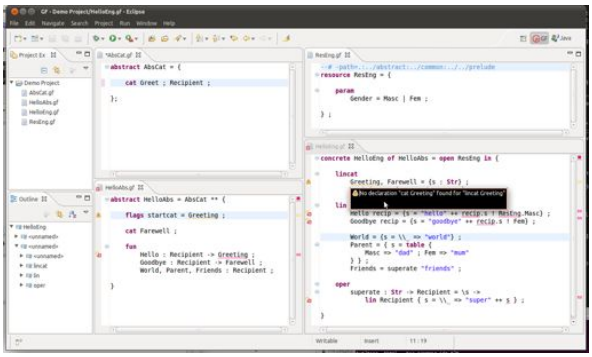
GETTING STARTED

1. Create a new blank General project in the usual way. If asked whether you want to add the Xtext nature to your project, you can safely say no.
2. There is a wizard for adding a new GF [3] source file from File > New > Other > GF [3] Source File:

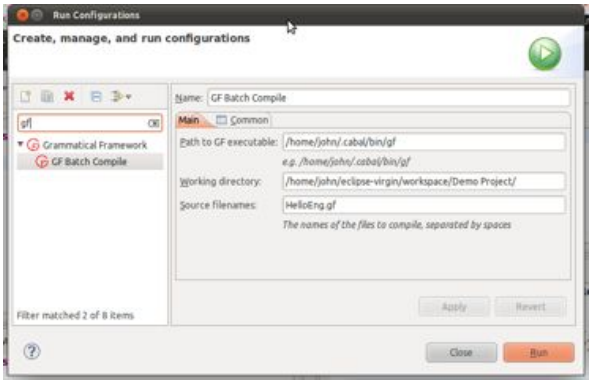


3. You can find a small example at http://www.grammaticalframework.org/eclipse/examples/hello/ [16]. Download the files and manually add them to your Eclipse workspace.

4. Note how changing a `cat` definition for example will produce warnings and/or errors in other the modules.



5. Compile your source using the provided Run Configuration.



KNOWN ISSUES

1. Local parameter/binding identifiers show up as unresolved, e.g. `recip` in:

```
Hello recip = {s = "hello" ++ recip.s ! Masc} ;
```

2. Qualified names currently aren't treated correctly, so `Masc` works but `ResEng.Masc` does not.
3. If the Apply button in the Run Configurations dialog doesn't remains disabled, swap to the Common tab and back.
4. Selective inheritance has not been properly tested.
5. Interfaces/functors are currently not supported.
6. The in-editor validation often needs to be triggered by some keystrokes, especially when Eclipse laods with some already-opened files.

5. Future direction: example-based grammar writing

It is typically the case that the writer of a GF [3] concrete grammar is at least fluent in the language and has GF [3] skills which are directly proportional with the complexity of the abstract syntax to implement. However, in the case of a rather complex multilingual grammar comprising 5 to more languages, as for instance was the case with the MOLTO [1] Phrasebook(reference...) which was first available in 14 languages and which has a reasonably rich semantic interlingua, the task of finding grammar developers is a difficult one. Even if there exist such developers, their task can still be made easier, by trying to automate where possible, and alleviate over certain technicalities of GF [3] programming that would slow down the grammar development.

When writing a application grammar, one such problem would be to use the resource library in order to build generate text for a given language with the help of the primitives already defined in the correspondent resource grammar. For this, however, one needs to be familiar with the almost 300 existing functions, assuming that the domain writer is different than the resource grammar write, as it is often the case.

In order to make the users' task easier, an API is provided so that the domain grammar writer only needs to know the GF [3] categories and how they can be built from each other. This layer makes the interaction with the resource library smoother for users, and also makes it easier to make new constructions from the library available.

For example, the sentence "I talked to my friends about the book that I read", is parsed to the following abstract

syntax tree:

```
UseCl (TTAnt TPast ASimul) PPos (PredVP (UsePron i_Pron) (ComplSlash
(Slash3V3 talk_V3 (DetCN (DetQuant DefArt NumSg) (RelCN (UseN book_N)
(UseRCl (TTAnt TPast ASimul) PPos (RelSlash IdRP (SlashVP
(UsePron i_Pron) (SlashV2a read_V2)))))))
(DetCN (DetQuant (PossPron i_Pron) NumPl) (UseN friend_N))))
```

If we use the API constructors, the abstract syntax tree is simpler and more intuitive:

```
mkS pastTense (mkCl (mkNP i_Pron) (mkVP (mkVPSlash talk_V3 (mkNP the_Art
(mkCN (mkCN book_N) (mkRCl pastTense  (mkRCl which_RP (mkClSlash
(mkNP i_Pron) (mkVPSlash read_V2)))))))) (mkNP (mkQuant i_Pron)
plNum friend_N)))
```

In this way, the domain grammar writer, can just use the functions from the API, and combine them with lexical terms from dictionaries and functions from outside the core resource library that implement non-standard grammatical phenomena, that do not occur in all languages.

One step further in the direction of automating the development of domain grammars is to have the possibility to enter function linearizations as a positive example of their usage. This is particularly helpful in larger grammars containing syntactically complicated examples that would challenge even the more experienced grammarians. If instead an example is provided, even though the grammar could return more than one parse tree, the user can select the good tree or take advantage of the probabilistic ranking and take the most likely one.

The example-based grammar writing system is still work in progress, but there is a basic prototype of it available already, and it will be further developed and improved. The basic steps of the system will be shortly described further on, along with the directions for future work.

The typical scenario is a grammarian working on a domain concrete grammar for a given language - which we call X for convenience.

In this case, he would need at least a resource grammar for X. Preferably there should also be a large lexical dictionary and/or a larger-coverage GF [3] grammar with probabilities. Currently, larger lexical resources exist for English, Swedish, Bulgarian, Finnish and French. For Turkish there exists a large lexicon also, but the resource grammar is not complete.

We also assume that the user has an abstract syntax for the grammar already and that the _lincats_ (namely representations of the abstract categories in the concrete grammar) are basic syntactic categories from the resource grammar(NP, S, AP).

Consequently, the functions from the abstract syntax will be grouped in a topological order, where the ordering relation a < b <=> b takes a as argument in a non-recursive rule. There are no cycles in this chain of ordered elements, since a similar check is being performed at the compilation stage. The elements will be ordered in a list of lists - where every sub-list represents incomparable elements. The user will be provided first with the first sub-list and after completing it, with the next ones.

For each such function, an abstract tree from the domain grammar having as root the given function will be generated. The arguments are chosen among the functions already linearized. In case that another concrete grammar exists already, the user can also see a linearization of the tree in the other language, and also an example showing how the given construction fits into a context. For example, if the user needs to provide an example for Fish in a given grammar, say the tourist phrasebook, and there is an English grammar already then he would get a message asking him to translate fish as in "fish" / "This fish is delicious".

When providing the translation, the user will be made aware of the boundaries of the grammar, by the incremental parser of the resource grammar. If the example can be parsed and the number of parse trees is greater than 1, then either the user can pick the right one, or the system can choose the most probable tree as a linearization. From here, the system will also generalize the tree by abstracting over the arguments that the function could have. Finally the

resulting partial abstract syntax tree will be translated to an API tree and written as linearization for the given function.

The key idea is based on parsing, followed by compilation to API and provides considerable benefits,especially for idiomatic grammars such as the Phrasebook, where the abstract syntax trees are considerably different. For example, when asking for a person's name in English the question "What is your name" would be written using API functions as:

```
mkQCl (mkQCl whatSg_IP (mkVP (mkNP (mkQuant youSg_Pron) name_N)))
```

which stands for the abstract syntax tree:

```
UseQCl (TTAnt TPres ASimul) PPos (QuestVP whatSg_IP (UseComp
  (CompNP (DetCN (DetQuant (PossPron youSg_Pron) NumSg) (UseN name_N)))))
```

On the other hand, in French the question would be translated to "Comment t' appelles tu" (literally translated to "How do you call yourself") which is parsed to:

```
UseQCl (TTAnt TPres ASimul) PPos (QuestIAdv how_IAdv
  (PredVP (UsePron youSg_Pron) (UseV appeler_V)))
```

and corresponds to the following API abstract tree:

```
mkQS (mkQCl how_IAdv (mkCl p.name appeler_V)))
```

Currently, steps are made to integrate the system with the Web Editor and in this way combine the example-based methods with traditional grammar writing. In this case the set of functions that can be linearized from example will be computed incrementally, depending on the state of the code.

A similar procedure to the one that determines which functions can be linearized from example can be used to find the functions that can be tested - functions already linearized that can be learned from example. In this way, the functions linearized in the editor - manually or by example can be also tested by randomly generating an expression and linearizing it in the language that is under development and also in one or more languages for which a concrete grammar exists. In case the linearization is not correct, the user can proceed to ask for a new example, or to modify the linearization himself.

Other plans for future work, in addition to integrate the method with the GF [3] Web Editor, include a thorough evaluation of the utility of the method for larger grammars and with grammarians of different levels of GF [3] skills. Moreover, we plan to include a handle for unknown words, that should make it easier for the user to build a small lexicon from examples.

As a solution to this, we devised the example-based grammar learning system, that is meant to automate a significant part of the grammar writing process and ease grammar development.

The two main usages of the system are to reduce the amount of GF [3] programming necessary in developing a concrete grammar and the second and more important - to make possible learning certain features of a language for grammar development.

In the last years, the GF [3] community constantly increased and so did the number of languages from the resource library and the number of domain grammars using them. The writer of a concrete domain grammar is typically different than the writer of the resource grammar for the same language, has less GF [3] skills and is most likely unaware of the almost 300 constructors that the resource grammars implement for building various syntactical constructions; see http://www.grammaticalframework.org/lib/doc/synopsis.html [17].

6. Conclusion

We have presented two IDE's for GF [3]. The web-based IDE is a stable system, which makes it easy to develop multilingual applications in the cloud. The Eclipse plugin brings GF [3] to one of the leading desktop environments of software development. It is already usable for simple tasks such as syntax highlighting and cross-modular references, but more functionalities are being added; the further development of the Eclipse plugin will be sensitive to the actual users in the other sites of the MOLTO [1] project. In addition to the IDE's, we have introduced the technique of example-based grammar writing, which has already been implemented as a desktop shell program and within the web-based IDE.

The IDE's are expected to make the use of GF [3] more efficient for power users and more accessible for beginning users. The success in this will be monitored and evaluated in the case studies of the MOLTO [1] project.

documentation GF IDE RGL

Source URL: http://www.molto-project.eu/node/1379

Links:
[1] http://www.molto-project.eu
[2] http://www.molto-project.eu/University of Gothenburg
[3] http://www.grammaticalframework.org
[4] http://en.wikipedia.org/wiki/Integrated_development_environment
[5] http://www.eclipse.org/
[6] http://www.microsoft.com/visualstudio
[7] http://developer.apple.com/xcode/
[8] http://www.coderun.com
[9] http://www.grammaticalframework.org/
[10] http://www.grammaticalframework.org/demos/gfse/
[11] http://www.grammaticalframework.org/doc/gf-editor-modes.html
[12] http://www.grammaticalframework.org/compiler-api
[13] http://www.grammaticalframework.org/%7Ehallgren/Talks/GF/gf-ide.html
[14] http://www.eclipse.org/Xtext/
[15] http://www.grammaticalframework.org/eclipse/beta/
[16] http://www.grammaticalframework.org/eclipse/examples/hello/
[17] http://www.grammaticalframework.org/lib/doc/synopsis.html