1

An IDE for the Grammatical Framework

John J. Camilleri

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 libraries, and compilation and testing of grammars is greatly improved through single-click launch 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 entry to GF and encourage new users of the framework.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-247914.

1.1 Introduction

1.1.1 Grammatical Framework (GF)

GF is a special-purpose framework for writing multilingual grammars targeting multiple parallel languages simultaneously. It provides a functional programming language for declarative grammar writing, where each grammar is split between an abstract syntax common to all languages, and multiple language-dependent concrete syntaxes, which define how abstract syntax

A LaTeX Package for CSLI Collections.
Edie Tor and Ed Itor (eds.).

trees should be linearised into the target languages. From these grammar components, the GF compiler derives both a parser and a lineariser for each concrete language, enabling bi-directional translation between all language pairs. (Ranta, 2011)

Apart from being a standalone logical and natural language framework, there also exists an open-source collection of GF resource grammars for a number of natural languages, collectively known as the *Resource Grammar Library* (RGL) (Ranta, 2009). Currently comprising 24 natural languages from around the world, the libraries cover low-level syntactic features like word order and agreement in each particular language. These details are abstracted away from the application grammar developer through the RGL's common language-independent API, making it possible to write multilingual grammar applications without necessarily having any extensive linguistic training.

1.1.2 GF grammar development

As a grammar formalism, GF facilitates the writing of grammars which can form the basis of various kinds of rule-based machine translation applications. While it is common to focus on the theoretical capabilities and characteristics of such formalisms, it is also relevant to assess what software engineering tools exist to aid the grammar writers themselves. The process of writing a GF grammar may be constrained by the framework's formal limits, but its effectiveness and endurance as a language for grammar development is equally determined by the real-world tools which exist to support it.

Whether out of developer choice or merely lack of anything better, GF grammar development typically takes place in traditional text editors, which have no special support for GF apart from a few syntax highlighting schemes made available for certain popular editors[1]. Looking up library functions, grammar compilation and running of regression tests must all take place in separate windows, where the developer frequently enters console commands for searching within source files, loading the GF interpreter, and running some test set against a compiled grammar. GF developers in fact often end up writing their own script files for performing such tasks as a batch. Any syntax errors or compiler warnings generated in the process must be manually interpreted.

While some developers may actively choose this low-level approach, the number of integrated development environments (IDEs) available today indicate that there is also a big demand for advanced development setups which provide combined tools for code validation, navigation, refactoring, test suite management and more. Major IDEs such as Eclipse, Microsoft Visual Studio and Xcode have become staples for many developers who want more integrated experiences than the traditional text editor and console combination.

[1] See the GF Editor Modes page at `http://www.grammaticalframework.org/doc/gf-editor-modes.html`

1.1.3 Motivation

The goal of this work is to provide powerful development tools to the GF developer community, making more efficient the work of current grammar writers as well as promoting the Grammatical Framework itself and encouraging new developers to use the framework.

By building a GF development environment as a plugin to an existing IDE platform, we are able to obtain many useful code-editing features "for free". Thus rather than building generic development tools, we only need to focus on writing IDE customisations which are specific to GF, of course reducing the total effort required.

The rest of this paper is laid out as follows: section 1.2 describes the design choices which guided the plugin's development, section 1.3.1 then covers each of the major features provided by the plugin, and in section 1.4 we discuss our plans for evaluation along with some future directions for the work.

1.2 Design choices

1.2.1 Eclipse

Eclipse[2] is a multi-language software development environment which consists of both a standalone IDE, as well as an underlying platform with an extensible plugin system. Eclipse can also be used for the development of self-contained general purpose applications via its Rich Client Platform (RCP). The Eclipse Platform was chosen as the basis for a GF IDE for various reasons:

1. It is written in Java, meaning that the same compiled byte code can run on any platform for which there is a compatible virtual machine. This allows for maximum platform support while avoiding the effort required to maintain multiple versions of the product.
2. The platform is fully open-source under the Eclipse Public License (EPL)[3], is designed to be extensible and is very well documented.
3. Eclipse is a widely popular IDE and is already well-known to a number of developers within the GF community.
4. It has excellent facilities for building language development tools via the Xtext Framework (see below).

1.2.2 Xtext

Xtext[4] is an Eclipse-based framework for development of programming languages and domain specific languages (DSLs). Given a language description in the form of an EBNF grammar, it can provide all aspects of a complete language infrastructure, including a parser, linker and compiler or interpreter. These tools are completely integrated within the Eclipse IDE yet allow full customisation according to the developer's needs. Xtext can be used both for

[2] `http://www.eclipse.org/`
[3] `http://www.eclipse.org/legal/epl-v10.html`
[4] `http://www.eclipse.org/Xtext/`

creating new domain specific languages, as well as for creating a sophisticated Eclipse-based development environment.

By taking the grammar for the GF syntax as specified in Ranta (2011, appendix C.6.2), and converting it into a non-left recursive (LL(*)) equivalent, we used Xtext's ANTLR[5]-based code generator to obtain a basic infrastructure for the GF programming language, including a parser and serialiser. With this infrastructure as a starting point, a number of GF-specific customisations were written in order to provide support for linking across GF's module hierarchy system. Details of this implementation as well as other custom-built IDE features are described in section 1.3.1.

1.2.3 Design principles

Preserving existing projects

As users may wish to switch back and forth between a new IDE and their own traditional development setups, it was considered an important design principle to have the GF IDE *not* alter the developer's existing project structure. To this end, the GF Eclipse Plugin does not have any folder layout requirements, and never moves or alters a developer's files for its own purposes. For storing any IDE-specific preferences and intermediary files, meta-data directories are used which do not interfere with the original source files.

Preventing application tie-in in this way reduces the investment required for users who want to switch to using the new IDE, and ensures that developers retain full control over their GF projects. This is especially important for developers using version control systems, who would want to use the plugin without risking any changes to their repository's directory tree.

Interaction with GF compiler

It is clear that an IDE which provides syntax checking and cross-reference resolution is in some sense replicating the parsing and linking features of that language's compiler. With this comes the decision of what should be re-implemented within the GF IDE itself, and what should be delegated to the existing GF compiler. In terms of minimising effort required, the obvious option would be to rely on the compiler as much as possible. This would conveniently mean that any future changes to the language, as implemented in updates to the compiler, would require no change to the IDE itself.

However, building an IDE which depends entirely on an external program to handle all parsing and linking jobs on-the-fly is not a practical solution. Thanks to Xtext Framework's parser generator as described above, keeping all syntax checking within the IDE platform becomes a feasible option, in terms of effort required versus performance benefit. When it comes to reference resolution and linking however, it was decided that the IDE should delegate these tasks to the GF compiler in a background process (see section 1.3.4). This avoids the work of having to re-implement GF's module hierarchy system within the IDE implementation. Communication of scope

[5] `http://www.antlr.org/`

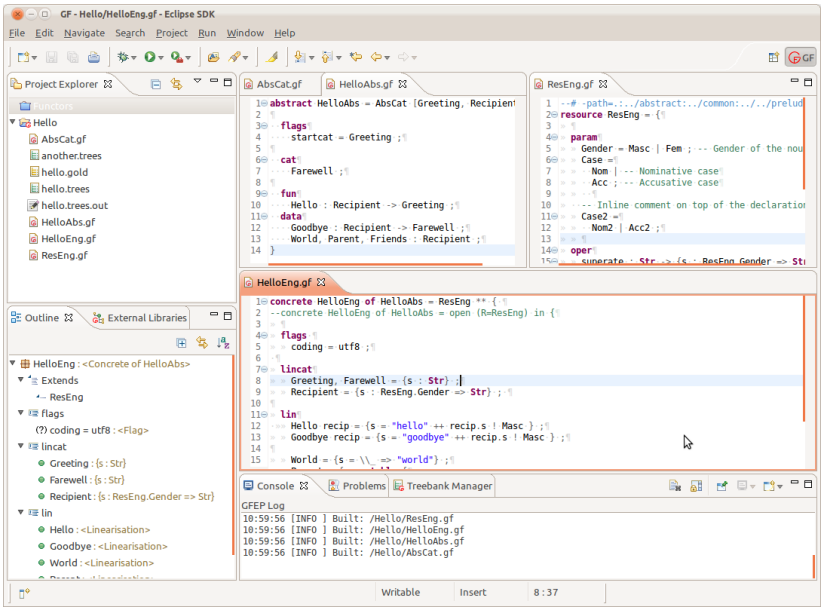

FIGURE 1 Screenshot of the GF Eclipse Plugin in use.

information from GF back to the IDE is facilitated through a new "tags" feature in the GF compiler, as described in section 1.3.3. This delegation occurs in a on-demand fashion, where the GF compiler is called asynchronously and as needed, when changes are made to a module's header.

1.3 The GF Eclipse Plugin (GFEP)

This section covers the major features provided by the plugin and their relevance to developers of GF grammars.

1.3.1 Code editing

Figure 1 shows a screenshot of the main IDE window. Note how multiple editor panes can be viewed simultaneously, by partitioning the workbench into arbitrary tabbed sections. Various source code-level features such as code folding, block-level indentation and commenting, and matching bracket highlighting are also provided. These basic code editing features, including the project navigation view in the top-left of the screen, are all provided directly by the Eclipse Platform.

Automatic formatting The built-in code formatter can be used to tidy one's code automatically, adhering it to the line break and indentation conventions as used in the GF book (Ranta, 2011). Figure 2 shows screenshots before and after invoking the code formatter.

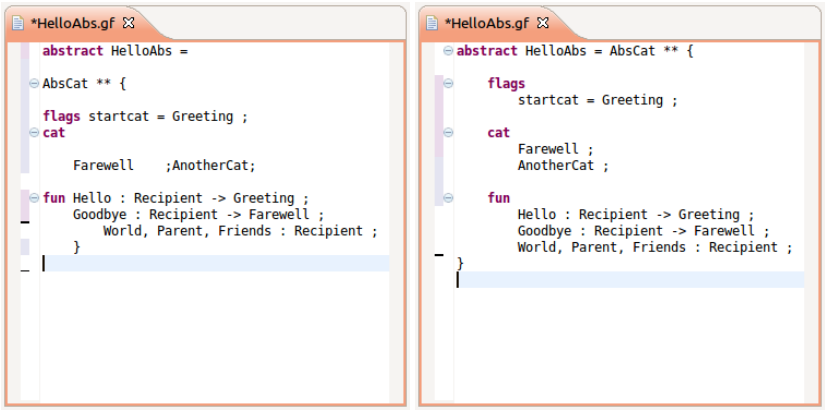

FIGURE 2 Before and after applying the automatic code formatting feature.

Wizards The plugin also provides some *wizards* for guiding developers in quickly creating new resources in the project, such as creating a new GF module from scratch, or cloning an existing module in one language into a new one.

Syntax validation As the basic language infrastructure for the IDE was generated from a grammar of the GF syntax, the plugin provides fully customisable syntax highlighting as well as instant syntax validation and marking of lexical errors. A variety of semantic warnings may also be shown to the user, for example indicating that a linearisation rule has no corresponding abstract function, or that an implemented interface has not been fully instantiated. Note that these features are all provided directly by the plugin implementation, without needing to call the standard GF compiler in the background.

Outline view The outline view in the bottom-left of figure 1 offers a complete overview of the current module structure. Every definition in the module is listed in a tree structure, along with its type information and helpful icons for quickly distinguishing the different judgement types. Clicking any of the terms will make make cursor jump to that point in the file, allowing for easy and quick navigation in large modules.

1.3.2 Launch configurations

Making use of Eclipse's launching framework, grammar writers have the ability to compile and run their modules with GF with a single click or button press. The plugin allows multiple *launch configurations* to be set up; each specifying the source modules to be compiled, any additional compiler flags, and any commands which should be passed to the GF shell for batch processing. Launch configurations can also be configured to automatically linearise treebank files for grammar regression testing (for more about this, refer to

section 1.3.5). The GF compiler can also optionally be launched into interactive shell mode, such that the user can interact with the GF interpreter in the traditional way without leaving the development environment.

Once set up, any launch configuration can be run quickly from within the IDE, avoiding the need to type in long terminal commands or scroll through one's shell history each time. A screenshot of the options available in the launch configuration dialog window is shown in figure 3.

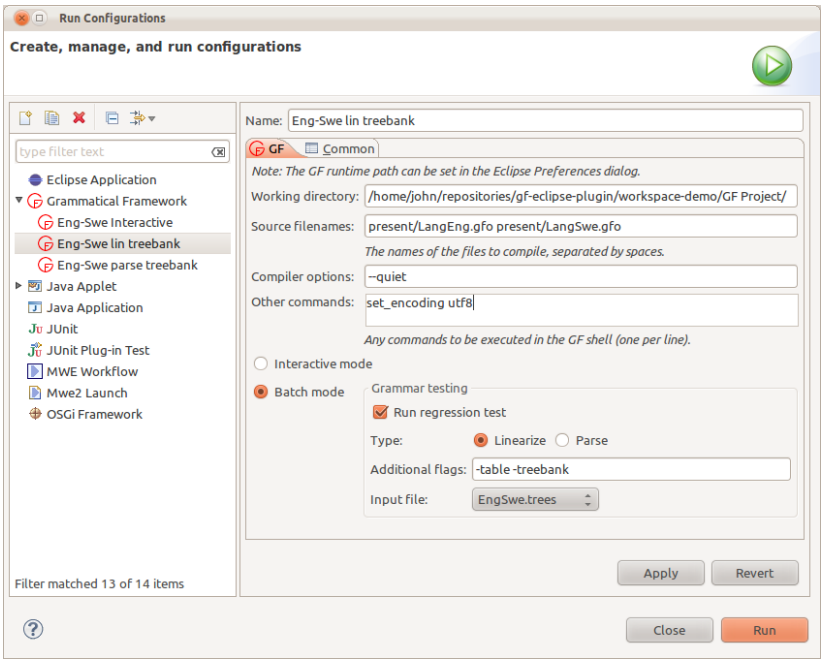

FIGURE 3 The launch configuration dialog, allowing developers to save their compilation flags and arguments for quick re-use.

1.3.3 Cross-reference resolution and scoping

As in most other programming languages, GF comes with a hierarchical module system which allows grammars to be split between multiple source files (modules), and for these modules to import and extend each other in an inheritable way. An identifier in a module which points to a function or value defined in another module is known as a *cross-reference*. The GF IDE must thus link all such cross-references between modules, allowing the developer to "jump" to their original points of definition, and indicate when a referenced identifier cannot be resolved.

A byproduct of this is the ability to display a list of all functions available in the module hierarchy, which are visible from any given point in a gram-

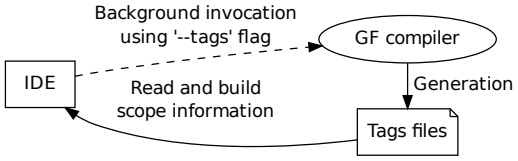

FIGURE 4 Tags files created by the GF compiler in a background process are used by the GF Eclipse Plugin for building scope information about the GF source files opened in the IDE.

mar. This is provided as an auto-completion pop-up dialog, which filters the displayed list of available functions by the characters preceding the current cursor position.

All this is achieved through the scoping infrastructure of the GF Eclipse Plugin, which can quickly find all visible definitions (i.e. the scope) for any part of a grammar. As this scope calculation is highly specific to GF's module system and inheritance syntax, rather than attempting to re-implement this behaviour within the IDE, it was decided that this task should be handled by the standard GF compiler system. The delegation of this work from the IDE to the GF compiler is handled by a custom Eclipse builder (see section 1.3.4). In order to facilitate communication between the IDE and the standalone compiler, a new tags-generation feature was added to GF. This is described in the following section.

GF tags generation

Tags files are used as a means of providing module scope information to the IDE from the GF compiler, when the latter is invoked as a background process via the GFEP automatic builder as depicted in figure 4. The tags generation in GF is inspired by popular tools like Ctags and Etags[6]. As of GF version 3.3.3, running the compiler with the `-tags` flag will begin the regular compilation pipeline, starting with the usual phases for parsing and analysing of the grammar code but stopping before any actual code generation. Instead, the compiler will write a set of `.gf-tags` files (one for each `.gf` source module) containing lists of every identifier in the scope of the current module. These files are saved in a tab-delimited format with one identifier per line, as shown in figure 5.

The first two fields of each line indicate the identifier name and the *kind* of declaration; that is, the keyword that is used for introducing the identifier, i.e. `fun`, `cat`, `lin`, `lincat` or `oper`. If the identifier is defined in the current module, then the third field contains the path to the source file along with

[6] `http://ctags.sourceforge.net/ctags.html`

```
mkN3  oper-def   .../ParadigmsEng.gf:406
mkN3  oper-type  .../ParadigmsEng.gf:118  {s : Number => Case => Str;...
mkPN  overload-def   .../ParadigmsEng.gf:390-393
mkPN  overload-type  .../ParadigmsEng.gf:390-393  Str -> {s : Case =>...
mkPN  overload-type  .../ParadigmsEng.gf:390-393  {s : Number => Case...
mkNoun  indir  ResEng  R  ResEng.gf-tags
```

FIGURE 5 Example of the `.gf-tags` file format, for the resource grammar library module `ParadigmsEng.gf` (some lines truncated for brevity).

the line number(s) for the definition. When the type is either `fun`, `oper` or `overload` then the final field contains the type signature for the identifier.

In addition to the declaration kinds listed above, the kind could also be specified as `indir`, which indicates that the identifier is imported from some other module, and that its definition should be looked up there. In this case, the following fields on the same line respectively contain the module name and alias under which the identifier was imported (where applicable), and the path to the `.gf-tags` file which contains the actual definition of the identifier. This is exemplified in the final line of figure 5 (`mkNoun`).

1.3.4 Automatic builder

The reliance on the GF compiler for providing scoping information means that repeated calls to this external program must be made by the IDE. This is handled by a custom Eclipse *builder*, which listens for changes in the project workspace, analyses the resource deltas and calls the GF compiler to refresh the scoping information. This generally happens each time a file is saved, however the plugin also attempts to detect when changes to the current module may have effects on its dependents, in which case it will update the scoping information for these descendants also. To reduce the total number of calls to the builder, the scoping information is only refreshed when changes are made to the module's *header* information.

In addition to obtaining scoping information as described in section 1.3.3 above, calling the GF compiler as a background task also allows any type errors not caught by the IDE directly to still be relayed back to the user. Since all GF grammars written in the IDE will ultimately have to be compiled with GF, it is important that all errors are bubbled up to the developer as soon as possible so that they do not go undetected for long.

1.3.5 Test case manager

As described in Ranta (2011, section 10.5), the typically recommended development-test cycle for GF grammars is as follows:

1. Create a file `test.trees` which contains a list of abstract syntax trees (one per line) to be tested.
2. Compile the grammar and linearise each tree to all forms, using a command such as:

   ```
   rf -lines -tree -file=test.trees | l -table -treebank
   ```

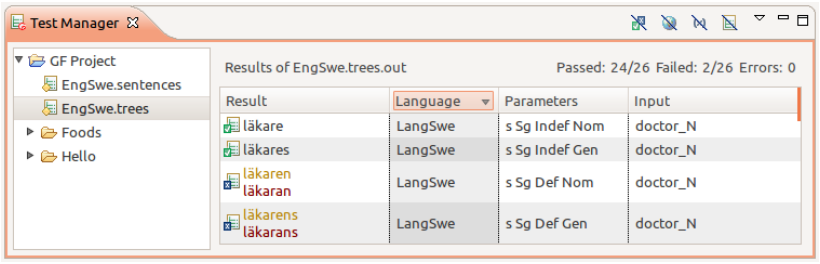

FIGURE 6 Viewing regression test output in the Test Manager view. In this example we see that the input tree `doctor_N` is incorrectly linearised as *läkaran* and *läkarans* for the singular definite cases in Swedish. The correct forms are *läkaren* and *läkarens*, respectively.

and capture the output in a file `test.trees.out`.

3. Manually correct the output in `test.trees.out` and save it as your gold standard file `test.trees.gold`.
4. Each time the grammar is updated, repeat step 2 and compare the new output against the gold standard using Unix `diff` or some other comparison tool.
5. Extend the tree set and gold standard file for every new implemented function.

The Tree Manager view in the GF Eclipse plugin provides a convenient graphical interface for managing this treebank testing process. This feature works together with the launch configurations to make the process of running grammar regression tests and gold standard comparisons quick and easy. As shown in the left-hand side of figure 6, all valid test input files in the project are shown together, and a simple double click on any will invoke the GF compiler, linearise the trees with the current version of the grammar and present comparisons against the corresponding gold standard in the right-hand panel. Various options are available for sorting and filtering the test results given, so that developers can quickly locate in which cases their grammar is failing.

Parsing While grammar testing is often focused on the linearisation of abstract syntax trees, the same procedure can be used equally as effectively for testing the *parsing* performance of the grammar under development. In this case, one would use `.sentences` instead of `.trees` files, containing plain-text sentences instead of abstract syntax trees, and the gold standard and output files would conversely contain the parse trees produced by the grammar.

1.4 Conclusions

Based on the Eclipse Platform an the Xtext framework, we have built a development environment for GF to replace the standard text editor and console window combination. While the GF Eclipse Plugin is not any more powerful in a computational sense, it does make available a number of devel-

opment tools and user interfaces for speeding up the writing and testing of GF grammars, as well as the use of existing resource libraries in application grammars.

The GF Eclipse Plugin is a new tool for the GF community, and as such its popularity and ability to increase grammar writing productivity remain to be seen.

1.4.1 IDE use and evaluation

Inevitably, it will often be the case that seasoned GF developers are happy with their current development environments and would be unwilling to switch to a new IDE-based setup. As a result, such developers are not considered the primary target for users of the GF Eclipse Plugin. Rather, the expected target group would be those developers who are already familiar with Eclipse or at least some similar IDE platform, even if they are not necessarily experienced in GF.

For this reason, plans are underway for an objective evaluation of the plugin to be carried out by a private company who already work in Eclipse but are new to GF. The experience of these developers with the new IDE will provide valuable information about the effectiveness of the GF plugin, where the normal learning curve for Eclipse itself will not be an issue.

1.4.2 Future work

Apart from optimisations in performance and addressing the issues already identified with the plugin to date, the following two major directions for future work have been identified.

Refactoring tools A highly useful component of many IDEs—which is currently missing from the GF Eclipse Plugin—is the availability of source code refactoring tools. Such tools could include generic refactoring tasks such as renaming identifiers (both locally and across modules) and moving function definitions, to more GF-specific ones such as extracting functors from groups of concrete syntaxes. Such tools have the potential to minimise time spent on repetitive programming tasks, minimise human error and indirectly promote adherence to coding conventions.

Source module API In order to perform syntax checking and module scoping the plugin must build internal models of a GF module's source code using the Eclipse Modelling Framework (EMF) and the derived language infrastructure as described in section 1.2.2. These models are only used internally and not exposed via any API. However, having this level of access to GF modules could open up many interesting possibilities, including graphical tools for grammar writing and integration with ontology management software. Implementing such an interface to the plugin's inner modelling information is certainly possible, although the effort required could only be justified if an appreciable demand for such a feature was expressed.

1.4.3 Availability

The GF Eclipse Plugin is freely available and any be used for any purpose. It is open source and released under the GNU General Public License (GPL)[7] (note that Xtext and the Eclipse Platform are covered by the Eclipse Public License[8]).

The official GF Eclipse Plugin web page[9] contains installation instructions, a user guide and tutorial screencast, the plugin's release history and links to the project's source code repository and issue tracker.

References

Ranta, Aarne. 2009. The GF resource grammar library. *Linguistic Issues in Language Technology* 2(2).

Ranta, Aarne. 2011. *Grammatical Framework: Programming with Multilingual Grammars*. Stanford: CSLI Publications. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

The Eclipse Foundation. 2011. Xtext 2.1 documentation. `http://www.eclipse.org/Xtext/documentation/2_1_0/Xtext%202.1%20Documentation.pdf`. Accessed March 2012.

[7] `http://www.gnu.org/licenses/gpl-3.0.txt`
[8] `http://www.eclipse.org/legal/epl-v10.html`
[9] `http://www.grammaticalframework.org/eclipse/`