# LWC'14: Rascal Submission

Tijs van der Storm
Centrum Wiskunde & Informatica
storm@cwi.nl / @tvdstorm

Pablo Inostroza Valdera
Centrum Wiskunde & Informatica
pvaldera@cwi.nl

January 2, 2014

## 1 The Rascal Language Workbench

Rascal is a meta programming language designed for transforming and analyzing source code in the broad sense [3] (http://www.rascal-mpl.org). Embedded in Eclipse, Rascal provides a versatile meta-environment for the design and implementation of domain-specific language parsers, analyzers, compilers and IDEs. Notable features include:

- Functional programming at the core, including support for higher-order functions, powerful pattern matching, comprehensions and extensible modules.
- Built-in context-free grammars backed by general parsing. As a result, there is no restriction on grammar rules (e.g., left-recursion is supported) and syntax definitions can be built in a modular fashion.
- Rewrite rules and traversal primitives for tree transformation (model-to-model). String templates for lightweight model-to-text transformation.
- Programmable IDE support based on Eclipse IMP [2], which provides out-of-the-box support for syntax highlighting, code folding, identifier hyper linking, hover documentation, outlining, error marking and builders.

## 2 QL and QLS base assignment

The implementation of QL and QLS consists of syntax definition and various analyses and transformations to implement semantic aspects such as type checking, normalization, and compilation. The implementation will be similar to earlier implementations of QL and QLS in Rascal [6, 5], but for LWC'14 it will generate code compatible with the reference solution [1].

The syntax of both languages is defined using Rascal's built-in context-free grammars. Annotations in the grammar enable custom syntax highlighting and code folding behavior in the QL/QLS editors. Semantic analysis and compilation are transformations of the syntax tree. Semantic analysis consists of name analysis (finding a declaration for every used question name) and type checking (ensuring that question are correctly used in expressions, and that conditions are boolean expressions). In both cases, the tree is decorated with errors and warnings (if any) which are shown at the appropriate locations in the QL/QLS editors. When type checking QLS models, the corresponding QL model is loaded first, to allow checking the consistency of question placement and widget assignment.

Compilation (code generation) consists of one or two phases, depending on whether QLS is used. In any case, QL models are transformed to an intermediate model. Then, this model is directly transformed to JavaScript code using Rascal's string templates. Or, alternatively, the intermediate model is first post-processed by the QLS compiler which, based on a QLS model, reorders questions, changes widget types, and adds styling information. After styling, the updated intermediate model is rendered to JavaScript code just like before.

# 3 Scalability and teamwork

**Scalability** Performance is an ongoing area of improvement in the Rascal implementation. Currently, Rascal is fully interpreted which could be a problem for performance. However, active development of the compiler for Rascal will significantly improve future performance. We hope to show the new compiler at the LWC'14 workshop.

The built-in parser of Rascal is based on GLL [4], an algorithm with a worst-case bound of $O(n^3)$. However, on average case grammars it is known to often perform much better. For instance, it is fast enough for interactive use in Rascal source code editors. Ongoing research at CWI aims to improve the performance of the parser as well.

**Teamwork** Since Rascal is a textual language workbench, any traditional version control system can be used to facilitate teamwork, on both language and questionnaires. The user of QL/QLS can use the built-in support in Eclipse for, e.g., Git, to show differences between files, apply patches, or commit changes to version control.

Nevertheless, because the QL/QLS DSL is a high-level and declarative language, it is easy provide additional support. Along this line of thought, the Rascal implementation will contain semantic impact analysis of changes to QL/QLS models. This will allow the modeler to inspect the semantic effects of a particular set of changes, in order to validate that an applied patch does not introduce logic that was unintended.

# 4 Demonstration at LWC'14

To demonstrate how well the Rascal implementation of QL scales, we will use the Binary Search Questionnaire suggested in the reference implementation [1]. Running Rascal's benchmarking library on parsing, parsing+checking, and parsing+checking+compilation on increasing sizes of this questionnaire, allows us to plot out the behavior of runtime performance of the various components.

With respect to teamwork, we assume that the traditional version-control work flow is well-known and does not need demonstration. Instead we will show our approach of semantic impact analysis using a concrete example.

# References


[1] M. Boersma and A. Hulshout. Language workbench challenge 2014: Reference implementation. Online, November 2013. `https://github.com/dslmeinte/LWC2014`.

[2] P. Charles, R. M. Fuhrer, S. M. S. Jr., E. Duesterwald, and J. J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In *OOPSLA*, pages 191–206. ACM, 2009.

[3] P. Klint, T. van der Storm, and J. Vinju. Rascal: A domain-specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[4] E. Scott and A. Johnstone. GLL parse-tree generation. *Science of Computer Programming*, 78(10):1828–1844, 2013.

[5] T. van der Storm. Demoqles: QL and QLS in rascal. Online, July 2013. `https://github.com/cwi-swat/demoqles`.

[6] K. van der Vlist, J. van der Woning, and T. van der Storm. QL-R-Kemi: QL and QLS in rascal. Online, June 2013. `https://github.com/cwi-swat/QL-R-Kemi`.