Software evolution series 0

October 2025

1 Introduction

Read the installation instructions[1] and install Rascal. We advise you to use the VScode version with the 'auto update' feature disabled. After you have installed Rascal, you can create a new Rascal project within VScode as follows:[2]

- In VScode, press `Ctrl+Shift+p` and then search for `Create a Rascal terminal` and press enter. This should open a Rascal terminal for you.
- Within the Rascal terminal execute the following snippets:

  ```
  import util::Reflective;
  newRascalProject(|home:///my-project-name|);
  ```

  Of course, you can give a different location to the `newRascalProject` function. Read more about locations in the documentation: https://www.rascal-mpl.org/docs/Rascal/Expressions/Values/Location/

Opening a folder in VScode automatically creates a workspace. The VScode workspace is referred to in Rascal via the *project* location.

1.1 Loading SQL projects

On Canvas you can find two SQL-related test projects that we will be using throughout this course. (But feel encouraged to find open source projects as additional data source in your experiments.) To make these projects available for analysis from within Rascal, you need to add the projects to the VScode workspace. You can do this by going to

File → Add Folder to Workspace

To load the SQL projects within Rascal, you may need to change the name of the *.project* file of the project to *pom.xml*, so a *$ mv .project pom.xml* is all you need (e.g., within smallsql0.21_src).

[1] https://new.rascal-mpl.org/docs/GettingStarted/

[2] Based on the following documentation: https://www.rascal-mpl.org/docs/GettingStarted/CreateNewProject/

2 Analyzing projects

Analyzing Java projects with Rascal is relatively simple, we just need the following two imports

```
import lang::java::m3::Core;
import lang::java::m3::AST;
```

For ease of use, you can also import the following packages:

```
import IO;
import List;
import Set;
import String;
```

See the corresponding documentation page for an overview of the exported functions.

Now we can create a function to walk over all files in the projects and parse their ASTs:

```
list[Declaration] getASTs(loc projectLocation) {
    M3 model = createM3FromMavenProject(projectLocation);
    list[Declaration] asts = [createAstFromFile(f, true)
        | f <- files(model.containment), isCompilationUnit(f)];
    return asts;
}
```

When an SQL project is added to your VScode workspace, you can call the above function as follows to load the SQL project as ASTs.

```
getASTs(|project://smallsql0.21_src/|);
```

And assign it to a variable in the REPL as follows.

```
asts = getASTs(|project://smallsql0.21_src/|);
```

Using the ASTs, we can analyze the Java project. For instance, we can write the following function to count the number of interfaces in the project:

```
int getNumberOfInterfaces(list[Declaration] asts){
    int interfaces = 0;
    visit(asts){
        case \interface(_, _, _, _, _, _): interfaces += 1;
    }
    return interfaces;
}
```

You can also extract information out of the AST nodes using the visit statement, which is needed for the assignments. So, figure out how to extract information from AST nodes. Also, what is the effect of using the \ before the node in a visit statement?

3 Assignments

3.1 Problem 1.

Create a Rascal function that calculates the number of for-loops in smallsql. You can find the type for Java ASTs here: https://www.rascal-mpl.org/docs/Library/lang/java/m3/AST/#lang-java-m3-AST-Declaration.

```
int getNumberOfForLoops(list[Declaration] asts){
    // TODO: Create this function
}
```

3.2 Problem 2.

Create a Rascal function that outputs most occurring variable(s) and how often they occur:

```
tuple[int, list[str]] mostOccurringVariables(list[Declaration] asts){
    // TODO: Create this function
}
```

3.3 Problem 3.

Create a Rascal function that outputs most occurring number literal(s) and how often they occur:

```
tuple[int, list[str]] mostOccurringNumber(list[Declaration] asts){
    // TODO: Create this function
}
```

3.4 Problem 4.

Create a Rascal function that outputs the locations where null is returned:

```
list[loc] findNullReturned(list[Declaration] asts){
    // TODO: Create this function
}
```

4 Rascal tests

Rascal has built-in functionality to work with tests, this is achieved by using the *test* modifier for a function:

```
test bool numberOfInterfaces() {
    return getNumberOfInterfaces(getASTs(|project://smallsql0.21_src|)) == 1;
}
```

In the Rascal REPL, you can now run *:test* and Rascal will run the tests of your project. For now, test functions can **not** have arguments.