Software evolution series 0

October 2022

1 Introduction

Read the installation instructions[1] and install Rascal. We advise you to use the VScode version.

1.1 Creating a Rascal project

The Rascal VScode plugin expects a certain project structure to work correctly. To set this up, we provide a small script that you can find on Canvas named *createRascalProject.sh*. Execute it as follows: *./createRascalProject.sh projectName*, where you replace *projectName* with your preferred project name. This script creates a valid Rascal project in the current working directory. Now, you can open the project with VScode.

1.2 VScode workspaces and Rascal projects

Opening a folder in VScode automatically creates a workspace. The VScode workspace is referred to in Rascal via the *project* location.[2] Therefore, to make other projects available to this URI form in the Rascal VScode plugin, you need to add it to the VScode workspace. You can do this by going to

File → Add Folder to Workspace

, and selecting the correct folder. Now you can add both SQL projects to the current workspace.

To be able to load the SQL projects with Rascal, you need to change the name of the *.project* file to *pom.xml*, so a *$ mv .project pom.xml* in the project directory is all you need.

[1] `https://new.rascal-mpl.org/docs/GettingStarted/`
[2] `https://new.rascal-mpl.org/docs/rascal/expressions/values/location/`

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;
}
```

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:

```
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]] mostOccurringVariable(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.