CWI Centrum Wiskunde & Informatica

Hack your DSL with Rascal: Exercises

Tijs van der Storm, Pablo Inostroza Valdera, CWI

Part I

Before starting coding, make sure you have opened a Rascal console associated with the project `RascalQLTutorial` (right-click on any Rascal file in the project and select 'Start console'). Then, in the console, do:

- `import exercises::ImportThis;`
- `import exercises::Snippets;`
- past statements from *exercises/Snippets.rsc* and see what happens.

The exercises can be complete by directly editing *exercises/Part1.rsc* and *exercises/Part2.rsc*.

0. FizzBuzz

(See http://c2.com/cgi/wiki?FizzBuzzTest)

Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print "FizzBuzz".

Tips

- [1..101] gives the list [1,2,3,...,100]
- use `println` to print.

1. Adding *unless*

Add an unless statement which is to be used similar to `ifThen` statements:

```
unless (x > 1) { "What is your age?" age: int }
```

- add a production to **Question** in *QL.rsc*
- add a constructor to **Question** in *AST.rsc*
- add a tc rule to the type checker in *Check.rsc*
- add a normalize rule to the normalize in *Normalize.rsc* (NB: the semantics of `unless(e, s)` is equivalent to `if(not(e), s)`)

Check in the IDE that the type checker indeed signals errors in unless conditions and bodies, and that its conditions appear in the outliner.

Tip

- implement `unless` analogous to `ifThen` in all cases

Optional Exercises

a. change the typechecker so that a warning is issued in the case of `ifThen(not(_), ...)`.
b. fix the outliner (*Outline.rsc*) so that `unless` conditions appears in the outline.
c. fix the formatter (*Format.rsc*)to pretty print `unless`.

2. Date valued questions

Add support for date valued questions:

- add syntax to **QType** to allow date fields (*Lexical.rsc*)
- add new **QType** constructor for date types (*AST.rsc*)
- add new case to **type2widget** in *Compile.rsc* to generate DateValueWidgets (see *resources/js/framework/value-widgets.js*)

3. Conditional expressions

Add conditional expression `x ? y : z`

- add production to Expr (*QL.rsc*)
    - Make sure it's low in the priority hierarchy i.e. `x && y ? a : b` should be parsed as `(x && y) ? a : b`.
- add new `Expr` constructor in *AST.rsc*
- add new case to `typeOf` in *TypeOf.rsc*
- add new case to `tc` in *CheckExpr.rsc*
- add new case to `expr2js` in *Expr2JS.rsc*

Part II

4. Explicit desugaring of *unless* to *ifThen* using `visit`

Warm up

I. use `visit` print out all labels in a form
II. use `visit` count all questions (question/computed)

Explicit desugaring of `unless`:

- use `visit` to traverse and rewrite the Form

- use pattern matching to match on unless nodes.
- rewrite unless nodes to `ifThen` using `=>`

The desugar function is called before compilation so the compiler (*Compile.rsc*) does not have to be changed to support `unless`, even if no `normalize()` was used.

Tip

- See examples of `visit` in *Resolve.rsc* and *Outline.rsc*

Optional

a. add `unlessElse`, and desugar it to `ifThenElse`.
b. write a transformation using `visit` to simplify algebraic expressions (e.g., `1 * x`, `0 + x`, `true && x`, `false && x`, etc.).

5. Extract data dependencies

Warm up

I. use deep matching (using `/`) to find all variables (`Id`) in a form.
II. use deep match to find all question with label value (within the quotes) equal to name; make sure there are such labels in your test code.

A computed question is dependent on the questions it refers to in its expression. Such dependencies can be represented as a binary relation (a set of tuples). The goal of this exercise is to extract such a relation.

- use the `Node` and `Deps` types and `nodeFor` function shown in (*Dependencies.rsc*)
- visit the form, and when encountering a computed question record edges to the `Deps` graph to record a data dependency.
- use deep match (`/`) to find `Id`s in expressions

Tips

- check out examples of deep match in *Compile.rsc* and *Check.rsc*
- have a look at `controlDeps`, defined in (*Dependencies.rsc*) for inspiration
- use the function `visualize(Deps)` (*Visualize.rsc*) to visualize the result of your data dependency graph. Click on nodes to see the location they correspond to.

Part III

6. Implement a Rename refactoring.

In this exercise we will employ concrete syntax matching and transformation to define a rename refactoring for QL. It's important for refactorings to preserve as much existing layout as possible. Hence, refactorings typically cannot be implemented in terms of abstract syntax, because it would require pretty printing the transformed code.

Implementing a rename refactoring proceeds in two phases:

- Compute all occurrences corresponding to a certain name; that is, its definition and all the references to it.
- Syntactically transform the program so that all occurrences corresponding to a name are consistently renamed.

Optional

a. Think about name consistency preconditions before you can apply a rename refactoring safely
b. Extend the refactoring invocation in `Plugin.rsc` so that an error message is shown if the precondition does not hold.

7. Check for use before define of questions

Although QL allows the use of a question in some expression before it actually appears in the form, one could say that this represents a kind of code smell. In this exercise you will define an analysis that checks for this smell.

- use the resolved relation of Exercise 5 to find the the order required by the dependencies (use the order() function analysis::graphs::Graph to compute topological order).
- determine textual ordering by comparing the .offset field of locs

Tips

- check out examples of deep match in *Compile.rsc* and *Check.rsc*
- Use the resolved relation of Exercise 5 to find the the order required by the dependencies (use the `order()` function `analysis::graphs::Graph` to compute topological order).
- Determine textual ordering by comparing the `.offset` field of source locations.

Optional

a. Hook up the analysis to the type checker so that warnings are shown in the editor when a question is used before it's defined.

8. Reorder questions to eliminate used-before-define questions

Whereas Exercise 6 was concerned with identifying a code smell, in this exercise you will write a refactoring to eliminate the smell, namely by reordering questions so that no use-before-define questions are present.

This is best illustrated using an example:

```
"q1" q1: int = 2 * q2
"q2" q2: int
```

Should be transformed to:

```
"q2" q2: int
"q1" q1: int = 2 * q2
```

Optional

a. Think about how to eliminate the code smell with as little change as possible, and implement that.