# CREATING CUSTOM RULE PRIMITIVES FOR JENA

HENRIETTE HARMSE

In this post I will show you

1. how to add your own custom rule primitive,
2. how to inform Jena of your custom rule primitive, and
3. I will discuss things you have to keep in mind when writing a custom primitive.

## 1. Adding a Custom Rule Primitive

A powerful feature of Jena is that it allows you to create your own custom builtin rule primitives. Building on our student example of the previous post, assume we want to calculate the final mark for a student given their test result, exam result and project result. We assume we have the following data

```
:Peet :hasTestResult 77 .
:Peet :hasExamResult 86 .
:Peet :hasProjectResult 91 .
```

for which we add the following rule

```
[calcStudentFinalMarkRule:
  (?student :hasTestResult ?testResult)
  (?student :hasExamResult ?examResult)
  (?student :hasProjectResult ?projectResult)
  calcFinalMark(?testResult, ?examResult, ?projectResult, ?finalMark)
    -> (?student :hasFinalMark ?finalMark)]
```

This states that when `?student` has a `?testResult`, `?examResult` and `?projectResult`, a `?finalMark` can be calculated for which we add an associated triple for `?student`. What is left for us is to implement our `calcFinalMark` primitive, which we do by creating a class `CalcFinalMark` that extends the `BaseBuiltin` abstract class. To name our custom primitive we override `getName()`:

```
@Override
public String getName() {
  return "calcFinalMark";
}
```

We state that our custom primitive has 4 parameters by overriding:

```
@Override
public int getArgLength() {
  return 4;
}
```

To ensure that `calcFinalMark` can be used both in the body and the head of rules we override

---

*Date*: 21st April 2018.

```
@Override
public boolean bodyCall(Node[] args, int length, RuleContext context) {
  return doUserRequiredAction(args, length, context);
}

@Override
public void headAction(Node[] args, int length, RuleContext context) {
  doUserRequiredAction(args, length, context);
}
```

The meat of the implementation is the `doUserRequiredAction(args, length, context)` method, which consists of the following steps:

1. check that we have the correct number of parameters,
2. retrieve the input parameters,
3. verify the typing of input parameters,
4. doing the actual calculation,
5. creating a node for the output parameter, and
6. binding the node to the output parameter.

```
private boolean doUserRequiredAction(Node[] args, int length,
  RuleContext context) {

  // Check we received the correct number of parameters
  checkArgs(length, context);

  boolean success = false;

  // Retrieve the input arguments
  Node studentTestResult = getArg(0, args, context);
  Node studentExamResult = getArg(1, args, context);
  Node studentProjectResult = getArg(2, args, context);

  // Verify the typing of the parameters
  if (studentTestResult.isLiteral() && studentExamResult.isLiteral() &&
    studentProjectResult.isLiteral()) {
    Node finalMark = null;
    if (studentTestResult.getLiteralValue() instanceof Number &&
      studentExamResult.getLiteralValue() instanceof Number &&
      studentProjectResult.getIndexingValue() instanceof Number) {

      Number nvStudentTestResult =
        (Number)studentTestResult.getLiteralValue();
      Number nvStudentExamResult =
        (Number)studentExamResult.getLiteralValue();
      Number nvStudentProjectResult =
        (Number)studentProjectResult.getLiteralValue();

      // Doing the calculation
      int nFinalMark = (nvStudentTestResult.intValue() * 20)/100 +
        (nvStudentExamResult.intValue() * 50)/100 +
        (nvStudentProjectResult.intValue() * 30)/100;

      // Creating a node for the output parameter
```

```
      finalMark = Util.makeIntNode(nFinalMark);

      // Binding the output parameter to the node
      BindingEnvironment env = context.getEnv();
      success = env.bind(args[3], finalMark);
    }
  }
  return success;
}
```

## 2. Registering a Custom Primitive with Jena

Our code for load our rules and activating it is similar to my previous post, except that you have to make a call to register the custom primitive:

```
// Load RDF data
String data = path.toFile().getAbsolutePath() +
  "/src/main/resources/data2.ttl";
Model model = ModelFactory.createDefaultModel();
model.read(data);

// Register custom primitive
BuiltinRegistry.theRegistry.register(new CalcFinalMark());

// Load rules
String rules = path.toFile().getAbsolutePath() +
  "/src/main/resources/student2.rules";
Reasoner reasoner = new GenericRuleReasoner(Rule.rulesFromURL(rules));

InfModel infModel = ModelFactory.createInfModel(reasoner, model);
infModel.rebind();
```

## 3. Things to Keep in Mind

There are two main things I think one needs to keep in mind with Jena custom rule primitives:

1. A primitive is suppose to be a elementary building block. Being able to create your own primitives may tempt you to add all sorts of interesting processing besides the manipulation of triples, but I strongly advice against that. Arbitrary processing in your builtin primitive can degrade performance of inferencing.
2. Do not assume that you have control over when a rule will be triggered. Exactly when a rule will be triggered is dependent on when the Jena `InfModel` implementation decides to re-evaluate the rules, which is dependent on internal caching of the `InfModel` implementation and how it deals with modifications made to the model. Even though I believe `InfModel` implementations will avoid arbitrarily re-evaluating rules, I still think it is conceivable that under some circumstance the same rule may be triggered more than once for the same data. Furthermore, the Jena documentation of 3.6.0 states that the `InfModel` interface is still in flux, which could mean that even if a rule is only triggered once for given data currently, due to unforeseen changes it may be triggered more than once in future updates of Jena.

## 4. Conclusion

In this post I gave an example of how you can develop a custom rule primitive for Jena. The code for this example can be found at github.

## References