How-To: New Code Generator Plugin

This short How-To describes the process to create a new code generator plugin for Gaalop.

Classes to write

Purpose	Description	Base Classes
Plugin Class	This class represents your Plugin to Gaalop. It contains its name and a description of its function. The most important method of your plugin class however is the factory method that creates new instances of your code generator class.	`de.gaalop.CodeGeneratorPlugin`
Code Generator	This class takes a control flow graph and returns source code. Internally you should use a Visitor to traverse the graph and accumulate the code in a buffer.	`de.gaalop.CodeGenerator`
Visitor	This class will traverse the control and dataflow graphs to generate source code for your code generator.	`de.gaalop.cfg.ControlFlowVisitor` `de.gaalop.dfg.ExpressionVisitor`

Dependencies

Your plugin should only depend on `api-1.0.0.jar`, which can be found in the Gaalop `plugins` directory.

Packaging

The classes of your plugin need to be packaged as a JAR archive. To make Gaalop aware of your plugin, you need to include a special file in your JAR archive.

Filename: META-INF/services/de.gaalop.CodeGeneratorPlugin
Content: One line that contains the fully qualified name of your Plugin Class (as described above).

Installation

To install your Plugin, you only need to put your JAR archive on the Gaalop class file. If you use the standard Gaalop starter, you can simply drop your JAR archive into the `plugins` folder in the Gaalop installation directory.

Miscellaneous Notes

You can implement the `ControlFlowVisitor` and `ExpressionVisitor` interface both in one class or split them up into two classes. This is up to your personal taste. Implementing both interfaces in one class is easier, but will make your class bigger.
To correctly add parenthesis around expressions and respect the operator priority of your target language, you should implement your own operator priority table. You can add a helper method to your visitor that checks the operator priority of a parent and child node and adds parenthesis as

needed. See the example project for an implementation of this.

If your target language does not support the outer and inner product of the geometric algebra, simply throw an `UnsupportedOperationException` in those two methods of your `ExpressionVisitor` implementation.

If your `CodeGenerator` implementation does not have any fields, you can implement it as a singleton and let the factory method in your Plugin return the singleton instance.

Your `CodeGenerator` and `CodeGeneratorPlugin` implementations must be thread-safe. If they are immutable this is not of great concern.

If you need a logging facility for your plugin, consider using Apache Commons Logging and Log4j since both are already included in the Gaalop distribution.

You can generate the filename of your output file(s) using the following helper method:

```
    private String generateFilename(ControlFlowGraph in) {
         String filename = "gaalop.ext";
         if (in.getSource() != null) {
             filename = in.getSource().getName();
             int lastDotIndex = filename.lastIndexOf('.');
             if (lastDotIndex != -1) {
                 filename = filename.substring(0, lastDotIndex);
             }
             filename += ".ext";
         }
         return filename;
     }
```

Simplify replace `ext` with a file extension of your liking.