Japanese Global AST Transformations

Groovy 1.6 offers offers several approaches to transforming the AST of code within the compiler. You can write a custom AST visitor, you can use annotations and a local AST transformation, or you can use a global AST transformation.

This page explains how to write and debug a global AST transformation.

Sticking with the naive and simple example from the local transformation page, consider wanting to provide console output at the start and stop of method calls within your code. The following "Hello World" example would actually print "Hello World" along with a start and stop message:

```
def greet() {
   println "Hello World"
}

greet()
```

Not a great use case, but it is useful to explain the mechanics of global transformations.

A global transformation requires four steps: 1) write an ASTTransformation subclass, 2) create a Jar metadata file containing the name of your ASTTransformation, 3) create a Jar containing the class and metadata, and 4) invoke groovyc with that Jar on your classpath.

Writing an ASTTransformation

This is almost exactly the same step you'll need if writing a local transformation. You must define an ASTTransformation subclass that reads, and possibly rewrites, the syntax tree of the compiling code. Here is the transformation that will add a console start message and end message to all method invocations:

```
@GroovyASTTransformation(phase=CompilePhase.CONVERSION)
public class LoggingASTTransformation implements ASTTransformation {

   static final def TARGET = WithLogging.getName()

   public void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
       List methods = sourceUnit.getAST()?.getMethods()
       methods?.each { MethodNode method ->
           Statement startMessage = createPrintlnAst("Starting $method.name")
           Statement endMessage = createPrintlnAst("Ending $method.name")

           List existingStatements = method.getCode().getStatements()
           existingStatements.add(0, startMessage)
           existingStatements.add(endMessage)
       }
   }

   private Statement createPrintlnAst(String message) {
       return new ExpressionStatement(
           new MethodCallExpression(
               new VariableExpression("this"),
               new ConstantExpression("println"),
               new ArgumentListExpression(
                   new ConstantExpression(message)
               )
           )
       )
   }
}
```

The first line (@GroovyASTTransformation) line tells the Groovy compiler that this is an AST transformation that should occur in the conversion CompilePhase. Unlike local transformations, global transformations can occur in any phase.

The publicvisit(ASTNode[], SourceUnit) method is invoked for each source unit compiled. In this example, I'm just pulling out all the methods defined in the source. A method to the compiler is simply a list of Statement objects, so I'm adding a statement zero logging the start message and appending a statement to the end of the list with the end message.

Notice the complexity in creating a simple println Statement in the createPrintlnAst method. A method call has a target(this), a name(println), and an argument list(the message). An easy way to create AST is to write the Groovy code you expect to create, then observe what AST the compiler generates within the IDE's debugger. This requires a test harness with a custom GroovyClassLoader and an AST Visitor.

Writing Jar Metadata

The Groovy compiler discovers your ASTTransformation through a file named "org.codehaus.groovy.transform.ASTTransformation". This file must contain the fully qualified package and name of your transformation. In my example, the file simply has one line:

```
gep.LoggingASTTransformation
```

Creating the Jar

The ASTtransformation and the metadata must be packaged into a single Jar file. The org.codehaus.groovy.transform.ASTTransformation file must be in the META-INF/services directory. The Jar layout for this example follows:

```
LogMethodTransform.jar
--gep
----LoggingASTTransformation.class
----LoggingASTTransformation$_visit_closure1.class
--META-INF
----services
------org.codehaus.groovy.transform.ASTTransformation
```

Compiling the Example

The new Jar must be put on the groovyc classpath for the transformation to be invoked. If the sample script at the top of the post is in a file named "LoggingExample.groovy", then the command line to compile this is:

```
groovyc -cp LogMethodTransform.jar LoggingExample.groovy
```

This generates a LoggingExample.class that, when run with Java, produces:

```
Starting greet
Hello World
Ending greet
```

Debugging Global Transformations

Local transformations are simple to debug: the IDE (at least IDEA) supports it with no extra effort. Global transformations are not so easy. To test this you might write a test harness that invoked LoggingASTTransformation on a file explicitly. The test harness source is available and could easily be modified to fit your needs. Let me know if you know an easier way to debug this!

Applying your transformation on custom file extensions

Since Groovy 1.7.5, you are allowed to add your own file types to the groovy compiler. This can be especially useful when you have written a DSL which requires extensive AST transformations and that you want to avoid your transformation to be applied on regular groovy files. For example, imagine your DSL specifies user stories :

```
story "user enters invalid postcode" {

  given ....

  expect ...

}
```

Then you could allow this syntax in .story files only. To do this, you must first have registered your global AST transformation into the '*META-INF/services/org.codehaus.groovy.transform.ASTTransformation*' file like explained before. The second step is to register your custom file extension into the '*META-INF/services/org.codehaus.groovy.source.Extensions*' file :

```
story
```

The last step is to update your AST transformation so that it checks the file extension :

```
void visit(ASTNode[] nodes, SourceUnit source) {
        if (!source.name.endsWith('.story')) {
            return
        }
        // start transforming the AST

    }
```

Now you have a global AST transformation that will only apply on .story files.