

Groovy AST Transformations

Who Am I?: David Clark

Contact Info: david@psionicwave.com

Presentation Information:

<https://github.com/dwclark/AstTransformations>

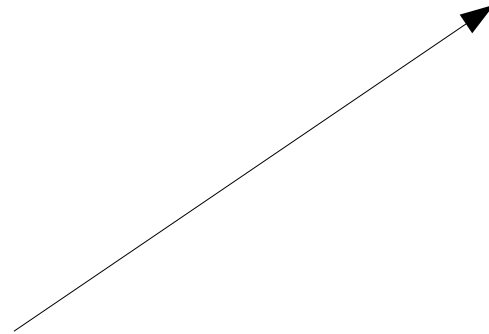
Notes:

See ast.odp/ast.pdf in git hub repo

How do you make
the perfect
logging statement
in Java?

What is Wrong Here?

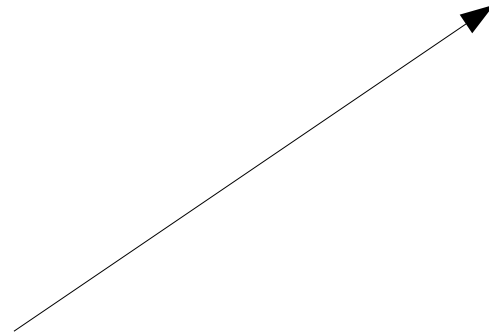
```
log.debug("Interesting stuff " + stuff);
```



This will result in a concatenation operation even if the debug level is not enabled

Fix #1

```
log.debug("Interesting Stuff {}", stuff);
```



This fixes the problem of concatenation always happening, `stuff` is passed to the `log.debug()`, and will only be evaluated if the debug method performs a log operation.

However, this only works if `stuff.toString()` does the “right thing.” So what do we do if `stuff.toString()` doesn't give us what we want?

Fix #2

```
if(log.isDebugEnabled()) {  
    log.debug("Interesting Stuff {}",  
        stuff.expensive());  
}
```

This fix is “correct” and does the “right thing.” `stuff.expensive()` is only called if the debug level is enabled. We could even go back to simple string concatenation and not use the template braces.

However, this is 1) much uglier, 2) Adds visual noise, and 3) introduces a new possible problem: If we decide to change this to log at a different level (say the warn level) then we have to remember to change it in both places. Anyone here ever guarded a warn logging statement with an `isDebugEnabled()` method call?

What's the Real Fix?

```
log.debug("Interesting stuff " + stuff);
```

Go back to the original version!

And get a real programming language!

The Real Problem is With the Language

- It's important to keep in mind that your logging library knows exactly when it should log something, this is not a problem of lack of information.
- But, the string concatenation will happen before your logging library can do something intelligent to stop it.
- The string template approach only works some of the time and solves the problem by exploiting a separate feature of the language, the fact that we can sometimes arrange things so that argument evaluation is cheap.
- But the only way to really solve the problem in java is to duplicate the logic for deciding when to debug. It's already inside the logging statements, now have to move it outside too.
- To solve the problem completely we need to change the semantics of Java. We need a way to tell it to only do the concatenation if the proper level is enabled (which Java knows).

Enter Groovy AST Transformations

What Does This Do?

```
@Log4j
public class NeedToLog {
    public void myMethod() {
        log.debug("Interesting Stuff " +
                  stuff);
    }
}
```

@Log4j tells the groovy compiler to load an AST transformation which:

Alters the NeedToLog class and gives it a private static final log variable and initializes it with the appropriate Logger.

Wraps all log statements inside the NeedToLog class with the appropriate guards. The log.debug() call will be wrapped inside an isEnabled() guard.

Technically groovy cheats a little and uses a ternary expression to make the transformation easier, but it does the same thing as the if guard.

Time For A Little Compiler Theory.

Don't worry, this shouldn't hurt (well that much anyway).
Cheating is encouraged.

What is an Abstract Syntax Tree (AST)?

Abstract syntax trees are a data structure widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

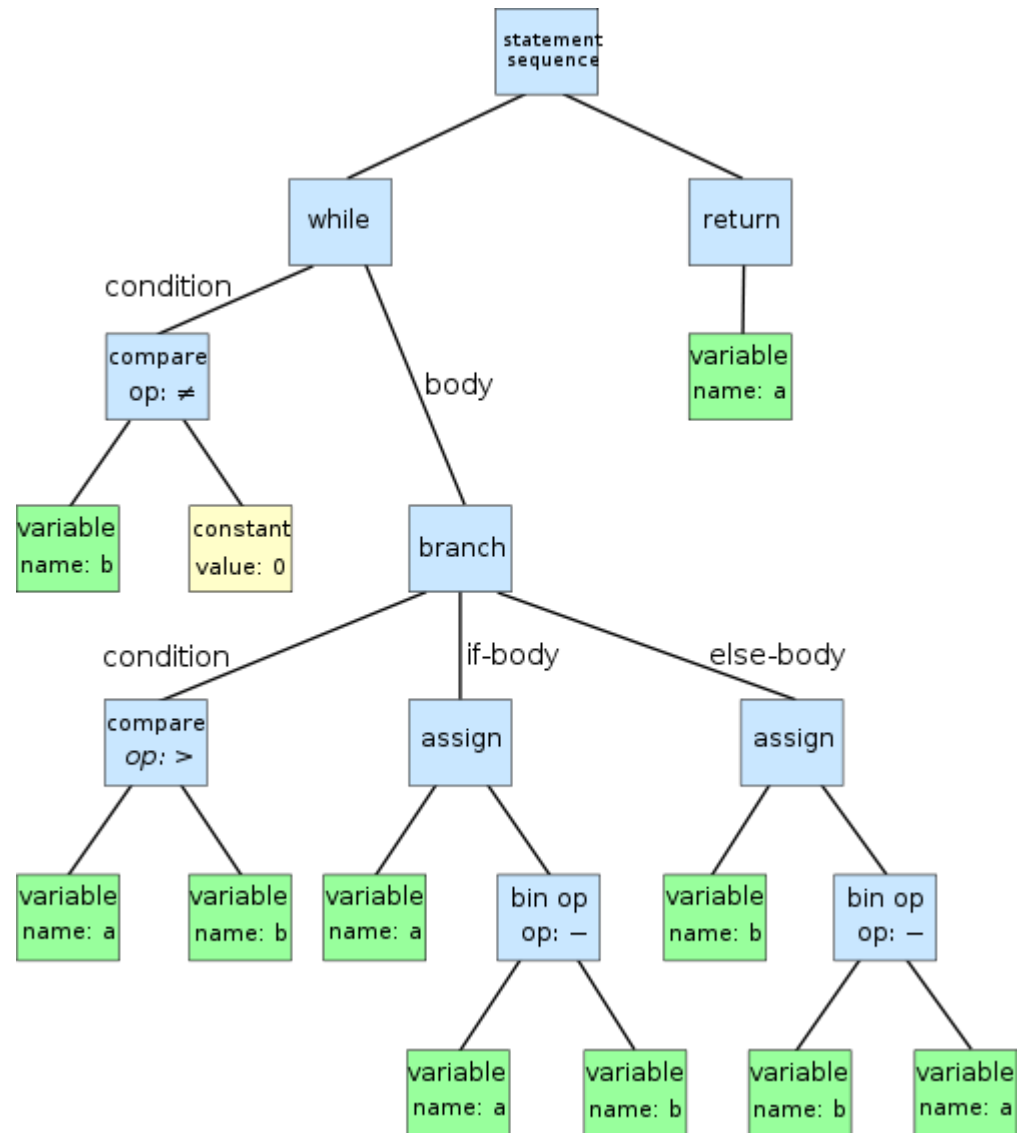
The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. Also, during semantic analysis the compiler generates the symbol tables based on the AST. A complete traversal of the tree allows a compiler to verify the correctness of the program.

After verifying the correctness, the AST serves as the base for the code generation step. It is often the case that the AST is used to generate the 'intermediate representation' '(IR)' for the code generation. (Wikipedia, *Abstract Syntax Tree*)

Take Home Message: The AST is a representation of your program using a tree data structure.

What does an AST look like?

```
while(b != 0) {  
    if(a > b) {  
        a = a - b;  
    }  
    else {  
        b = b - a;  
    }  
}  
  
return a;
```



A little more about ASTs

There are two main ways to represent ASTs inside a compiler:

Homogeneous AST: With a homogeneous AST all nodes in the tree are the same type/class. This makes it very easy to walk the tree and the AST itself ends up looking like a plain vanilla tree. The disadvantage of this approach is in storing and using information specific to each node type. For example, a class declaration supplies information that is very different than an if statement, but in a homogeneous AST they are both represented by the same node type.

Heterogeneous AST: All nodes are subclasses of a base node. This makes it harder to traverse a tree, but very easy to store node specific data. Here the AST looks less like a tree and more like objects containing other objects.

Groovy uses a heterogeneous AST. To make walking the tree easier, the Groovy libraries supply classes which implement the visitor pattern for every node type used by Groovy. We'll see an example of this soon.

Some Final Bits of Compiler Theory

Syntax: In simplest terms, the syntax is what the program looks like.

Semantics: What the program means.

For example, suppose I have two functions that compute the area of a triangle, one written in lisp, the other in java. The semantics of both functions is the same, they mean the same thing. However, they will look very different, in other words the syntax is different.

An AST transformation changes the semantics of a program not its syntax. In fact an AST transformation cannot change the syntax of groovy. If a program is not syntactically valid, your AST transformation will never be invoked. All AST transformations are run after the syntax is checked.

This means you cannot introduce new syntax using an AST transformation. Whatever you want to do must be expressed using regular groovy syntax.

However, your transformations can change the meaning of the valid code as much as you want. For example you could write an AST transformation to make all '+' operations mean subtraction and all '-' operations to mean addition. This is not a good idea, but it is a simple example of the kind of power an AST transformation has.

How Does Groovy See This?

```
def foo = 5 + 5;
```

As a BlockStatement which contains a single Statement

That single statement is an ExpressionStatement

That ExpressionStatement wraps a DeclarationExpression

That DeclarationExpression has a VariableExpression on the left side

And that DeclarationStatement has a BinaryExpression on the right side

That BinaryExpression contains two ConstantExpressions and a '+' Token

Duh? Right? What you call yourselves programmers?

Don't worry, I cheated to figure this out, and you can cheat too.

Groovy Console, The Cheating AST Programmer's Best Friend

- ▼ ClassNode - script1375322813893
 - ▶ Constructors
 - ▼ Methods
 - ▶ MethodNode - main
 - ▼ MethodNode - run
 - ▼ BlockStatement - (1)
 - ▼ ExpressionStatement - DeclarationExpression
 - ▼ Declaration - (foo = (5 + 5))
 - Variable - foo : java.lang.Object
 - ▼ Binary - (5 + 5)
 - Constant - 5 : int
 - Constant - 5 : int
 - ▶ MethodNode - this\$dist\$invoke\$3
 - ▶ MethodNode - this\$dist\$set\$3
 - ▶ MethodNode - this\$dist\$get\$3
 - ▶ MethodNode - <clinit>

Note that the natural representation for this is as a tree.

The Basic Idea For Writing Any Groovy AST Transformation

- 1) Write the code that you would write by hand if you were not writing an AST transformation (i.e. write source code, not AST nodes).
- 2) Load the script into the Groovy Console and see how Groovy makes your source code into an AST.
- 3) Using the groovy AST packages build up AST nodes in code. This is nothing more than regular object oriented programming. The only difference is that you are modelling source code, not business requirements.
- 4) Run the AST transformation inside Groovy Console and analyze both 1) the source code that your AST produces and 2) The actual AST that your code produces.
- 5) Repeat steps 3 and 4 iteratively and converge onto your solution
- 6) Profit!!

Example #1: Writing Methods With AST Transformations

Motivation: I often want two methods that do the exact same thing except that one of them runs synchronously and the other asynchronously.

The Wrong Solution: Write two methods. One that does the normal synchronous stuff and a second that invokes the first method from inside a Runnable or Callable.

Why Is This The Wrong Solution?: I don't want to test two methods. It is tedious and error prone. The asynchronous method is something I SHOULD test since I wrote it, but testing asynchronous stuff is difficult.

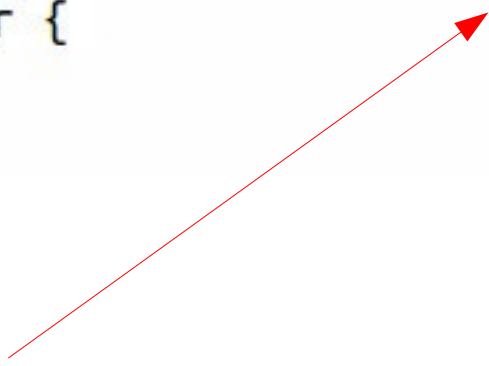
The Right Solution: Get the compiler to write the asynchronous method for me. The compiler already knows the method name and signatures. It knows where the threading stuff is located. As a bonus, if the signatures of the method changes, the asynchronous method will change too. If the synchronous method goes away, so does the asynchronous method.

Source Files: `transformations/MakeAsyncPair.groovy`,
`transformations/MakeAsyncPairTransformation.groovy`, `demo/ShowAsynPair.groovy`

Source Code File #1: The Java Annotation

```
import org.codehaus.groovy.transform.GroovyASTTransformationClass
import java.lang.annotation.* 1) This is a regular Java Annotation

@Retention(RetentionPolicy.SOURCE) 2) No need for runtime retention
@Target([ElementType.METHOD])
@GroovyASTTransformationClass (classes=[MakeAsyncPairTransformation])
public @interface MakeAsyncPair {
    String value() default "";
}
```



3) When the groovy compiler finds this annotation invoke the specified transformation, passing it the annotated element. In this case it will be a method node

Source Code File #2: The AST Transformation

1) Standard imports for an AST Transformation

```
import org.codehaus.groovy.control.CompilePhase;
import org.codehaus.groovy.transform.*;
import org.codehaus.groovy.ast.*
import org.codehaus.groovy.ast.expr.*
import org.codehaus.groovy.ast.stmt.*
import org.codehaus.groovy.control.SourceUnit;
import static org.objectweb.asm.Opcodes.*;
import org.codehaus.groovy.syntax.*;
```

2) There are 9 compiler phases, CANONICALIZATION is a good default.

```
@GroovyASTTransformation(phase = CompilePhase.CANONICALIZATION)
public class MakeAsyncPairTransformation implements ASTTransformation {
```

3) Implement ASTTransformation, it has a single method, visit.

```
    void visit(ASTNode[] astNodes, SourceUnit sourceUnit) {
        //do stuff here

        AstTransformUtils.fixupScopes(sourceUnit);
    }
}
```

4) Magic happens here.
Without this all kinds of weird errors pop up. Trust me, just stick it at the end of visit().

Source Code File #3:

Use The Annotation

```
@MakeAsyncPair("executorService")
public int addNumbers(int one, int two, int three) {
    return one + two + three;
}
```

More Compiler Stuff

Keep cheating and use the groovy api docs.

Classes

AnnotationConstantExpression
ArgumentListExpression
ArrayExpression
AttributeExpression
BinaryExpression
BitwiseNegationExpression
BooleanExpression
CastExpression
ClassExpression
ClosureExpression
ClosureListExpression
ConstantExpression
ConstructorCallExpression
DeclarationExpression
ElvisOperatorExpression
EmptyExpression
Expression
FieldExpression
GStringExpression
ListExpression
MapEntryExpression
MapExpression
MethodCallExpression
MethodPointerExpression
NamedArgumentListExpression
NotExpression
PostfixExpression
PrefixExpression
PropertyExpression
RangeExpression
SpreadExpression
SpreadMapExpression
StaticMethodCallExpression
TernaryExpression
TupleExpression
UnaryMinusExpression
UnaryPlusExpression
VariableExpression

Groovy Expr Package

Located At: `org.codehaus.groovy.ast.expr.*`

Expressions are things that evaluate to a value. However, this does not include the return statement, which is a `Statement` (get it?).

Most Expression objects are used by simply passing the correct arguments into their constructor. Sometimes you need to set a property after construction, but this is not as common.

You tend to build these expressions from the inside out. For example when building a binary expression, you tend to build the left side and right side, then build the actual expression itself.

Again, use the Groovy Console to explore actual syntax trees and then use that as a guide in creating the correct expressions.

Interfaces*LoopingStatement***Classes**

AssertStatement

BlockStatement

BreakStatement

CaseStatement

CatchStatement

ContinueStatement

DoWhileStatement

EmptyStatement

ExpressionStatement

ForStatement

IfStatement

ReturnStatement

Statement

SwitchStatement

SynchronizedStatement

ThrowStatement

TryCatchStatement

WhileStatement

Groovy Stmt Package

Located At: `org.codehaus.groovy.ast.stmt.*`

Statements are things which for the most part do not evaluate to a value but are still part of the program logic, they don't organize code.

Most Statement objects are used by simply passing the correct arguments into their constructor. Sometimes you need to set a property after construction, but this is not as common.

You tend to build these statements from the inside out.

`ExpressionStatement` is the bridge between statements and expressions. If something needs a statement, but you need to use an expression, wrap the expression using `ExpressionStatement`.

Again, use the Groovy Console to explore actual syntax trees and then use that as a guide in creating the correct expressions.

Interfaces

GroovyClassVisitor
GroovyCodeVisitor
Variable

Classes

AnnotatedNode
AnnotationNode
ASTNode
AstToTextHelper
ClassCodeExpressionTransformer
ClassCodeVisitorSupport
ClassHelper
ClassNode
CodeVisitorSupport
CompileUnit
ConstructorNode
DynamicVariable
EnumConstantClassNode
FieldNode
GenericsType
ImportNode
InnerClassNode
InterfaceHelperClassNode
MethodNode
MixinASTTransformation
MixinNode
ModuleNode
PackageNode
Parameter
PropertyNode
VariableScope

Groovy Ast Package

Located At: `org.codehaus.groovy.ast.*`

Nodes which are not expressions nor statements. These tend to be nodes which represent code organization.

Most ast node objects are used by simply passing the correct arguments into their constructor. Sometimes you need to set a property after construction, but this is not as common.

You tend to build these statements from the inside out.

Again, use the Groovy Console to explore actual syntax trees and then use that as a guide in creating the correct expressions.

ClassHelper is particularly useful because it helps in creating nodes that represent types (String, int, Date, etc.)

Example #2: Transforming Class Code with AST Transformations

Motivation: Sometimes I wish groovy structures behaved a little bit differently.

The Wrong Solution: Accept your fate. After all, it MOSTLY does what I want it to.

Why Is This The Wrong Solution?: Code should do what you mean, the less you change your thinking to fit the code, the more likely it is you will write bug free code. Plus, if you accept your fate you will never achieve Sith Lord status.

The Right Solution: Write and AST Transformation which changes the semantics (what the code means) but not the syntax (what the code looks like) of groovy.

Source Files: `transformations/EnableUnless.groovy`,
`transformations/EnableUnlessTransformation.groovy`, `demo/UseUnless.groovy`

Source Code File: Using ClassCodeVisitorSupport

```
class UnlessTransformer extends ClassCodeVisitorSupport {
```

```
    @Override
```

```
    public void visitMethod(MethodNode methodNode) {
```

```
        currentMethodNode = methodNode;
```

```
        super.visitMethod(methodNode);
```

```
    }
```

1) Override the correct methods to visit each node type you are interested in.

```
    @Override
```

```
    public void visitBlockStatement(BlockStatement block) {
```

```
        currentBlockStatement = block;
```

```
        super.visitBlockStatement(block);
```

```
    }
```

2) Be sure to always call the super method to make sure the code keeps walking.

```
    @Override
```

```
    public void visitExpressionStatement(ExpressionStatement expr) {
```

```
        currentExpressionStatement = expr;
```

```
        super.visitExpressionStatement(expr);
```

```
    }
```

3) Be careful when changing nodes, you can get concurrent modification exceptions. It is safer to replace something rather than removing/adding.

But Can't I Use AST Builders?

There is a builder which allows you to more concisely write AST transformations using more convenient builder syntax, or even simple text.

However, I don't recommend using it.

Problem #1: You tend to have to understand how to manually build nodes in order to make sense of the builder syntax.

Problem #2: The builders have limitations on what they can do.

Problem #3: You have much less control over how the AST is built when you are using builders.

Problem #4: They tend to not integrate well with plain AST nodes, which you usually have to end up using to fill in missing pieces of the builder.

While convenient, builder syntax has severe limitations at compile time. At compile time the groovy meta class machinery is not in operation, and much of the magic of builders comes from the meta class machinery.

Lesson: Stick with plain vanilla AST Nodes.

What Else Can I Do with AST Transformations?

I'm glad you asked.

Example #3: Optimize/Inline Code

Hotspot is great, but there is a limit to the miracles it can perform.

Method calls that are not inlined by hotspot have a definite overhead not just because it involves pushing/popping the stack, but it really messes up modern CPUs. Exotic optimizations such as deep pipelines, branch prediction, out of order execution are useless when you have to jump to code in another method.

However, Java has never had a way to force code inlining, but AST Transformations give you that power.

So what kind of optimization can you do with groovy AST Transformations?

To compare this I developed two identical numerical integration tests. The code they run is identical. Groovy Code: `transformations/Integrate.groovy`, `transformations/IntegrateTransformation.groovy`, `demo/ShowIntegrate.groovy`. Java code is in the `java` folder.

And the results are...

Example #3: The Results

Total # of floating point computations performed:

$$494,550 * 500 * 4 = 989,100,000$$

Time for Java to perform computations in a typical run:

30.035 seconds

Time for Groovy to perform computations in a typical run:

24.825 seconds

@Integrate AST + @CompileStatic + Indy speed advantage:

17.6% faster than java

For me this was the most interesting and fun result I found while preparing this presentation. Excuses that you can't use groovy because it is slow are just that, **excuses**.

Example #4: Static DB Code Generation

Keeping code in sync with a database is tedious, boring, and error prone.

You can do this with runtime metaclass magic, but the code tends to be slower (often this is irrelevant). However, this code cannot be directly used from java because java cannot see the groovy metaclass mojo.

Why not force the code to sync up with the database at compile time, statically write the correct properties and methods, and the java code will be able to consume this?

This has all of the advantages of traditional code generation but almost none of the drawbacks because you 1) never touch the code, 2) never see the code, and 3) the compiler never forgets to do the proper work at build time.

Examples: `sql/Person.groovy`, `sql/person.sql`, `transformations/DbClass.groovy`, `transformations/DbClassTransformation.groovy`.

I chuckled the first time I had to add the db library to the compiler's classpath.

Example #5: Code Removal

Sometimes you just want code to go away.

In C/C++ this is traditionally done with macros and `#defines`. Different compiler settings will use different `#defines` which will produce different code.

Groovyc allows for something nearly identical. You can configure the compiler to run arbitrary AST transforms at compile time, even if those annotations are not present on the classes or methods.

Suppose for example you want to be absolutely sure that there are no stray `println` calls in your web application. You can do this by executing an AST Transformation that silences all `println` method calls.

See `configscript/Noisy.groovy`, `configscript/config.gconfig`,
`transformations/NoPrintln.groovy`, `transformations/NoPrintlnTransformation.groovy`.

Use the Source Luke!

The best place to go to learn more about AST Transformations is the groovy source code. A lot of the newer features in the groovy 2.x series are AST Transformations.

`groovy.transform.*` defines the annotations for many of the groovy AST Transformations.

`groovy.lang.*` defines some older annotations for AST Transformations.

`org.codehaus.groovy.transform.*` defines the implementations for the AST transformations that come with groovy except for `@CompileStatic`.

`@CompileStatic` is quite extensive and gets its own package. It is in the `org.groovy.codehaus.groovy.transform.sc.*` package.

Final Thoughts

AST Transformations make groovy a complete programming language. Everything from the most static compile time trick to the most dynamic runtime technique is available in groovy.

The only other language I am familiar with that gives you this kind of flexibility and power is Common Lisp (Clojure being a lisp dialect may also give this kind of power).

Groovy AST Transformations are a bit more complicated than Common Lisp macros and are not quite as powerful. Lisp weenies (I'm including myself here) will still complain that Groovy can't compare to Common Lisp in generality and ease of use. But Groovy gets you a big chunk of what Common Lisp gives you and has the advantage of being used in production.

Questions?

Who Am I?: David Clark

Contact Info: david@psionicwave.com

Presentation Information:

<https://github.com/dwclark/AstTransformations>

Notes:

See ast.odt/ast.pdf in git hub repo