

GEP 2 - AST Builder Support

Metadata

Number:	GEP-2
Title:	AST Builder Support
Version:	7
Type:	Feature
Target:	1.7
Status:	Draft
Leader:	Hamlet D'Arcy
Created:	2009-04-01
Last modification:	2009-06-17

Abstract

Groovy 1.6 introduced the ability to perform local and global AST (Abstract Syntax Tree) transformations, allowing users to read and modify the AST of Groovy code as it is being compiled. Reading information in the AST is relatively easy in Groovy. The core library provides a strongly typed visitor called [GroovyCodeVisitor](#). Nodes can be read and modified using the API provided through the subtypes of [ASTNode](#). Writing new AST nodes is not as simple. The AST generated from source is not always obvious, and using constructor calls to generate trees of nodes can be verbose. This GEP proposes an ASTBuilder object that allows users to easily create AST.

The ASTBuilder object allows AST to be created from Strings containing Groovy source code, from a closure containing Groovy source, and from a closure containing an AST creation DSL. All three approaches share the same API: a builder object is instantiated and a build* method is invoked.

Approach

ASTNode from String

The simplest approach to implement is to provide an AST Builder with an API that takes a String and returns List<ASTNode>.

```

def builder = new AstBuilder()
List<ASTNode> statements =
builder.buildFromString(
    CompilePhase.CONVERSION,
    true,
    """ println "Hello World" """
)

```

- phase parameter tells the builder from which phase to return AST. This parameter is optional, and the default CompilePhase is CLASS_GENERATION. This provides a cleaner API for the common case, and CLASS_GENERATION was chosen because more types are available in later phases.
- the "statementsOnly" boolean parameter is an optional parameter, and tells the builder to discard the generated top level Script ClassNode. Default is true.
- The last String parameter is the input
- The builder returns List<ASTNode>

The above example produces the following AST:

```

BlockStatement
-> ExpressionStatement
    -> MethodCallExpression
        -> VariableExpression
        -> ConstantExpression
        -> ArgumentListExpression
            -> ConstantExpression

```

Alternatives

- Some sort of AST Template was considered for this feature. Consider the following example:

```

def astTemplate = builder.buildAst (
  "println $txt" ).head()

def constant = builder.buildAst ( "To be, or
not to be: that is the question" ).head()

def methodCallExpression =
  astTemplate.apply(txt: constant)
// method call expression not contains
println "To be ... "

```

This templating approach adds complexity that may not be used. It overloads the GString \$ operator, in that it is used here only with objects of type ASTNode but is used normally in GStrings with any Object type at all. Also, the templating approach can create order of precedence confusion. Consider source = "\$expr * y", and later \$expr is bound to "x+a". The result is "x + a * y", which was probably unintentional. At this time, the AST builder does **not** include such a feature.

ASTNode from Code Block

A useful API would be creating AST from code blocks.

```

AstBuilder builder = new AstBuilder()
def statementBlock = builder.buildFromCode
(CompilePhase.CONVERSION, true) {
  println "Hello World"
}

```

- Expressing Groovy source within Groovy source seems the most natural way to write it (as opposed to putting Groovy source into a String).
- Some IDE support is naturally available (highlighting, etc), but IDE warnings will be misleading for variable scoping rules
- Same issues and rules from "ASTNode from String" for phase and statementsOnly properties apply to this version
- Provides similar API as builder from String, except the code property accepts any block of code that is legal in the context of a Closure.
- Converting from a closure into AST is performed through a global compiler transformation. This requires that

the AstBuilder reference be strongly typed so that the global annotation can be triggered.

The above example produces the following AST:

BlockStatement

```
-> ExpressionStatement
  -> MethodCallExpression
    -> VariableExpression
    -> ConstantExpression
    -> ArgumentListExpression
      -> ConstantExpression
```

Alternatives

- If @ASTSource annotation is used, then it would be very easy to let users reuse that annotation outside of the builder. Consider the following example:

```
@AstSource(CompilePhase.CONVERSION)
List<ASTNode> source = { println "compiled
on: ${new Date()}" }
```

This option seems helpful; however, annotations on local variables are not yet supported. This approach will not be implemented.

ASTNode from psuedo-specification

Building AST conditionally, such as inserting an if-statement or looping, is not easily accomplished in the String or code based builders. Consider this example:

```

def builder = new AstBuilder()
List<ASTNode> statements =
builder.buildFromSpec {
    methodCall {
        variable "this"
        constant "println"
        argumentList {
            if (locale == "US") constant
"Hello"
            if (locale == "FR") constant
"Bonjour"
            else constant "Ni hao"
        }
    }
}

```

This library class is useful for several reasons:

- Using conditionals or looping within an AST Builder will probably be a common occurrence
- It is difficult to create a Field or Method references in any of the other approaches
- Simply using the @Newify annotation does not sufficiently improve the syntax
- This construct alleviates the need to distinguish between Statement and Expressions, since those words are dropped from the method names
- There is no need for a phase or statementsOnly property in this approach
- Many expressions take a type ClassNode, which wraps a Class. The syntax for ClassNode is to just pass a Class instance and the builder wraps it in a ClassNode automatically.

Issues

- Constructor parameter lists can be lengthy on ASTNode subtypes, and this approach removes the possibility for an IDE to help. This is the price to pay for a builder, and the planned builder metadata feature in 1.7 may alleviate this.
- The class creating AST from the psuedo-specification should be implemented so that it does not create a mirror-image class heirarchy of the current AST types. This would force all changes to the AST types to be performed in two places: once in the ASTNode subclass and once in this builder. If this is not possible, then at least the AST heirarchy doesn't change frequently.
- Several ASTNode types have constructor signatures all of the same type: (Expression, Expression, Expression) most commonly. This means the parameters in the DSL are order dependent, and specifying arguments in the wrong order doesn't create an exception but causes drastically different results at runtime. This is fully documented on the [mailing list](#).

- The syntax for specifying Parameter objects is documented on the [mailing list](#).
- A few of the ASTNode types having naming conflicts with language keywords. For instance the ClassExpression type cannot be abbreviated to 'class' and IfStatement cannot be reduced to 'if'. This is fully documented on the [mailing list](#).
- Parameters have default values and can be varargs. A suitable syntax needs to be proposed.
- Sometimes the order of the constructor parameters needed to be switched within the DSL. For instance, consider SwitchStatement(Expression expression, List<CaseStatement> caseStatements, Expression defaultStatement). The current syntax of the DSL imposes a sort of VarArgs rigidity on the arguments: lists are just implied by repeated elements. So having the middle parameter of SwitchStatement be a list is problematic because the natural way to convert the constructor is to have it become (Expression expression, CaseStatement... caseStatements, Expression default), which isn't possible. This is fully documented on the [mailing list](#).

Alternatives

Template Haskell and Boo provide a special syntax for AST building statements. Quasi-quote (or Oxford quotes) can be used to trigger an AST building operation:

```
ConstantExpression exp = [| "Hello World"
| ]
```

Those languages also supply a splice operator \$(...) to turn AST back into code. This is not part of the AstBuilder work.

References

Mailing-list discussions

- [\[groovy-user\] How to write an AST Builder](#)
- [Groovy AST Builder Discussion](#)
- [\[groovy-user\] Local Variables Declaration Discussion](#)
- [\[groovy-dev\] Several issues with GEP-2 AST Builder "from specification"](#)

JIRA issues

- This feature is dependent on allowing annotations on local variables - <http://jira.codehaus.org/browse/GROOVY-3481>

Useful links

- [Boo Meta Methods](#)
- [Template Haskell in Wikipedia](#)
- [Template Haskell Home Page](#)
- ["First Stab at Template Haskell" Blog Post](#)
- [Cython Metaprogramming Proposal](#) by Martin C Martin - Contains nice write up of use cases.

Reference Implementation

<http://code.assembla.com/AstBuilderPrototype> (very in-progress)

Test Case for AstBuilder from Code: <http://subversion.assembla.com/svn/AstBuilderPrototype/src/test/org/codehaus/groovy/ast/builder/AstFactoryFromCodeTest.groovy>

Test Case For AstBuilder from String: <http://subversion.assembla.com/svn/AstBuilderPrototype/src/test/org/codehaus/groovy/ast/builder/AstFactoryFromStringTest.groovy>

Test Case For AstBuilder from Specification: <http://subversion.assembla.com/svn/AstBuilderPrototype/src/test/org/codehaus/groovy/ast/builder/AstBuilderFromSpecificationTest.groovy><http://subversion.assembla.com/svn/AstBuilderPrototype/tests/org/codehaus/groovy/ast/builder/AstBuilderFromSpecificationTest.groovy>