

Groovy Compiler Metaprogramming and AST Transformations

Hamlet D'Arcy
Pearson

Who am I?

- * I write Java as a day job
- * Pearson was first full-time Java gig 3 years ago. On first day architect told me to learn Groovy.
- * Tried introducing more Groovy at work to mixed results
- * Prefer to write Groovy than Java in almost all cases

Why am I doing this

- * I like to write (-> my blog)
- * I like to meet people (-> user groups & conferences)
- * I like open source (Groovy contributor. Small contributor to JConch, Easyb, JetGroovy)

Traveling to Madison has exceeded my expectations!

What is Groovy?

"Groovy is the REAL Java 2. Henceforth, we can think of the Java language (not the platform) as forever 1.x. Groovy is what the real second version of Java should have been." Neal Ford (This is pretty questionable)

dynamic language for the JVM

- builds on strengths of Java
- inspired by Python, Ruby, Smalltalk
- almost-zero learning curve
- supports DSL Languages and compact syntax
- easy shell and build scripts
- reducing scaffolding code
- simplifies testing by supporting unit testing and more indirection
- seamlessly integrates with all existing Java objects and libraries
- compiles straight to Java bytecode so you can use it anywhere you use Java

Notes for those new to Groovy:

Looser syntax - No Parenthesis or semicolons

Looser typing – type annotations not required

Named Arguments – not just positional arguments

Closures

```
package example;

public class MyClass {

    private final List<String> list = new ArrayList<String>();

    public void add(String value) {
        try {
            list.add(value);
        } catch (Throwable t) {
            System.out.println(t);
        }
    }
}
```

Is this Java or Groovy?

This shows what in Groovy is the same as in Java:

- * keywords
- * exception handling
- * entity definitions and instantiation
- * Packaging & Imports
- * Java 5 enums, generics

“All Java programmers are already Groovy programmers”

What is different?

What's Different?

- * True OO with no primitives
- * Closures
- * Better syntax for new types
- * String templating
- * Operator overloading
- * Extended libraries

Groovy Console Demo (need nightly build of 1.7 Groovy Console)

Groovy Strings

```
def name = "Hamlet"
println "Hello $name"
```

Closures

```
3.times { println it }
(10..15).each { println it }
def x = [ 1, 2, 3]
def f = { List list -> list.each { println it } }
f x
```

Putting it all together

```
def names = ["Hamlet", "Dave", "Bucky"]
println ""
<root>
  <people>
    ${writer -> names.each { writer.write "  <person>${it}</person>\n" }}
  </people>
</root>
""
```

Amazing glue language. Mention optional static typing. Types get in your way for Controllers and components with no dependencies

How does it work?

groovy & groovyc - standard tools, work with Ant

Bidirectional compiler... and now Tri-directional compiler in 1.7!

It is Class files

Demo of scripts and javap output: (SimpleGroovyClass.groovy in scripts folder)

```
class Person {
  def firstName
  def lastName

  def greet() {
    println "Hello $firstName $lastName"
  }
}
```

Compiled from "SimpleGroovyClass.groovy"

```
public class Person extends java.lang.Object implements groovy.lang.GroovyObject{
  ...
  public Person();
  public java.lang.Object greet();
  protected groovy.lang.MetaClass $getStaticMetaClass();
  public groovy.lang.MetaClass getMetaClass();
  public void setMetaClass(groovy.lang.MetaClass);
  public java.lang.Object getFirstName();
  public void setFirstName(java.lang.Object);
  public java.lang.Object getLastName();
  public void setLastName(java.lang.Object);
  ...
}
```

What is Metaprogramming?

- writing a program that manipulates programs, including itself.
- Groovy makes metaprogramming easy and provides a number of techniques for doing so.
- It's **much more than open classes**.
- about creation and mutation of methods based on information not available at code time
- Allows more flexibility in your application, albeit at a potential increase in complexity

Types of things you can do:

- Intercept method calls to override/enhance the services an object provides
- Inject new methods to add to the list of services an object provides

Groovy is one of the best languages for runtime metaprogramming today.

Compile time metaprogramming is an alternative to runtime metaprogramming.

CTM is much more:

- * compile time static code analysis.
- * embedded languages
- * changing the semantics of the language.

It allows wild and different things to happen but is a bit of pandora's box.

We'll talk about

- What you can do
- How to do it
- and Walk through some code examples

We won't talk about: GORM, invokeMethod, or invokeProperty (much).

```
class Event {  
    @Delegate Date when  
    String title  
}
```

DelegateExample.groovy

```
class Event {  
    @Delegate Date when  
    String title  
}
```

```
% javap Event  
public class Event extends  
    ...  
    public long getTime();  
    public boolean after(java.util.Date);  
    public int getMonth();  
    ...
```

@Delegate is just a plain old annotation.
That means it is an interface with no implementation behind it.

Why does the bytecode have all the public methods of date?

Why doesn't Event extend Date or implement Date interface?


```
def shell = new ArithmeticShell()

assert 2 == shell.evaluate(' 1+1 ')
assert 1.0 == shell.evaluate('cos(2*PI)')

shouldFail(SecurityException) {
    shell.evaluate('new File()')
}
```

From “On Lisp”:

Lisp is an excellent language for writing compilers and interpreters, but it offers another way of defining a new language which is often more elegant and certainly much less work: to define the new language as a modification of Lisp. Then the parts of Lisp which can appear unchanged in the new language (e.g. arithmetic or I/O) can be used as is, and you only have to implement the parts which are different (e.g. control structure). A language implemented in this way is called an embedded language.

Spock Testing Framework

```
def "Does size method work?"() {  
  expect:  
    name.size() == size  
  where:  
    name << ['Madison', 'Dodgeville']  
    size  << [7, 10]  
}
```

Are those goto labels? (yes)

What error occurs when run this? (illegal forward references, uninitialized variables, int compared to List)

What really happens when you run this?

* where block executes first (where variables are defined)

* expect block executes second (where variables are parsed/used)

We'll come back to this later!

Groovy is a compiled language

...oh yes it is

Compiled changes visible in .class file

...visible to all JVM users

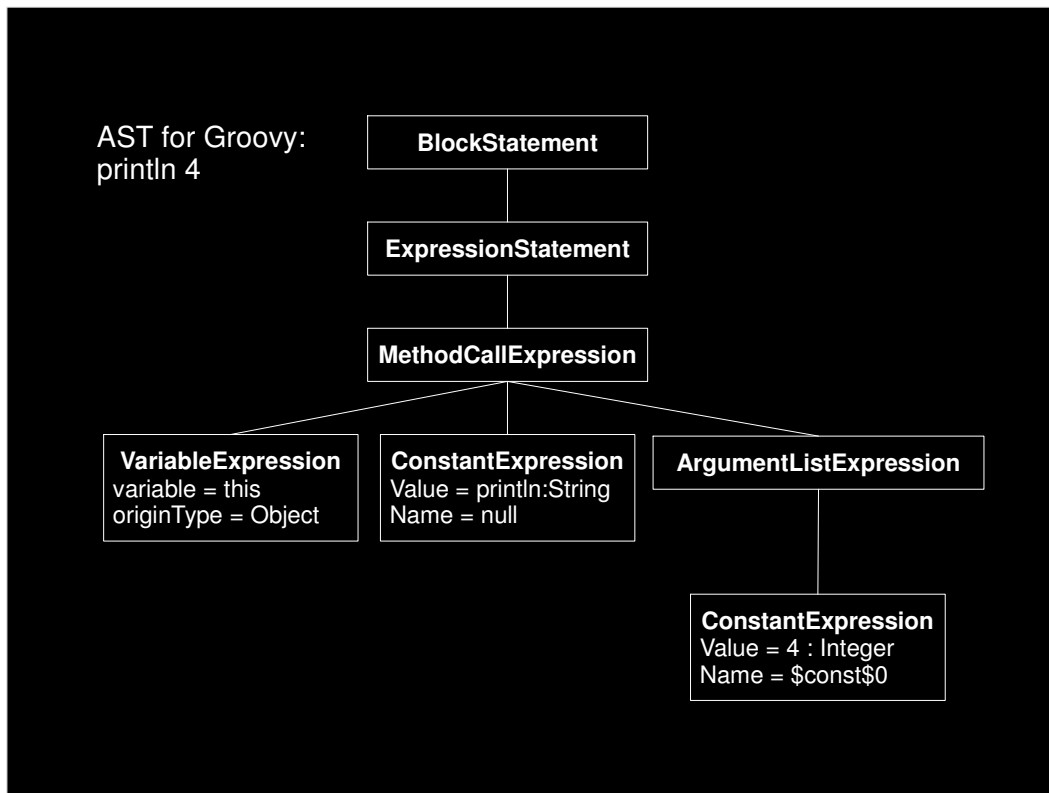
Language syntax is a library feature

...not hardcoded into the language

Groovy is compiled. It can be just-in-time compiled, but it is compiled and the bytecode is generated nonetheless.

Groovy language exposes several ways to hook into the compiler. You can see the code get transformed into an AST, and see that transformed into class output.

Anyone can do this. The semantics of the language is now a library feature you contribute to (although not the syntax).



Source is transformed into AST.

This is the AST for “println 4”

It is a tree structure and is typically walked with a Visitor. This pattern is implemented in Groovy.

- * Show GroovyConsole AST Viewer
- * Compare to Eclipse viewer
- * Compare to M/Oslo

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

Local AST Transformations =
Simple, Annotation Driven Alternative

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

```
groovysh> greet()
Starting greet
Hello World
Ending greet

groovysh>
```


(see code)

Local AST Transformations =
Simple, Annotation Driven Alternative

Annotation is used to *help* you find the AST insertion point and then you rewrite the AST somehow.

2 drawbacks: Locating splice point is not supported by libraries or language. Rewriting/producing AST is not supported by libraries/language (but AST Builder is coming soon).

```
cd scripts\localxform
groovyc *groovy
groovy -cp . LoggingExample.groovy
```



@Delegate	@Category
@Immutable	@Mixin
@Lazy	@PackageScope
@Newify	@Grab

Famous AST Transformations...

@Immutable – adds final + getters and no setters

@Lazy – adds lazy instantiation

@Newify – adds questionable ruby/python new syntax

@Category – easier category creation and usage

@Mixin – easier mixin creation and usage

@PackageScope – hack to add package scope to lang

@Grab – Great new dependency mgmt system

```
def shell = new ArithmeticShell()

assert 2 == shell.evaluate(' 1+1 ')
assert 1.0 == shell.evaluate('cos(2*PI)')

shouldFail(SecurityException) {
    shell.evaluate('new File()')
}
```

I wrote Arithmetic shell as an alternative to JEP, a arithmetic grammar and evaluator.

Provides advanced calculator like behavior to your application. Observes AST to restrict expressions to only those that are allowed in a calculator.

300 Line POGO alternative to custom grammar!
300 line alternative to expensive 3rd party library

Works off a visitor: each node type has a visit(ASTNode): void method.

Example: ./scripts/ArithmeticShell.groovy

Limitations: No real transforms b/c of visitor's void return type

Alternatives: 1) Finding all unbound variables in a script to generate cmd line help. 2) CodeNarc.

Spock Testing Framework

```
def "Does size method work?"() {  
  expect:  
    name.size() == size  
  where:  
    name << ['Madison', 'Dodgeville']  
    size  << [7, 10]  
}
```

- * Explain the path of execution
- * Show AST
- * Explain how it works

“Hijacking of goto” -> Drawback is you cannot change the grammar of the language. You're stuck with a Java-like grammar and most of the syntax. This can be very restricting.

Pitfalls!

Writing AST

Finding insertion points

Splicing AST into source

Splicing source into AST

Missing true transformations

Rigid Groovy/Java syntax

Variable capture

AstBuilder Coming in 1.7

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

AstBuilder is coming in Groovy 1.7 as part of GEP-2

Part of this is the AST DSL, which is a wrapper around all the constructors with some additional conveniences.

This example doesn't make this approach shine, but there is some merit in it.

AstFactory Coming in 1.7

```
def factory = new AstFactory()  
  
List<ASTNode> statements = factory.buildAst (  
    """ println "Hello World" """  
)
```

AstFactory “from String” coming in Groovy 1.7 (GEP-2)

Simple usage. Able to convert Strings of source into AST (you could easily pass file contents to it).

This is nice and immediately useful for newcomers.

There are parameters which aren't shown.

AstFactory Coming in 1.7

```
def factory = new AstFactory()

List<ASTNode> statements = factory.buildAst {
    println "Hello World"
}
```

AstFactory “from code” coming in Groovy 1.7 (GEP-2)

This is the holy grail of AST generation for 1.7

If this works, some feel the other two aren't needed!

It is quite hard to implement elegantly. Each ASTNode subtype needs special code and it feels like writing a quine each time. Ugh. This is the only approach not complete yet. Hopefully end of June timeframe.

Better Insertion Points in 2.0?

```
(defmacro nil! (var)  
  (list 'setq var nil))
```

```
(nil! x)
```

```
def x = ...  
nil! x  
assert !x
```

It is simple to rewrite a local variable at the calling location within a Lisp macro. `nil!` expands locally to set the value of `x` to null.

It is possible but not simple to do this Groovy. It is hard to find a method invocation called `nil!` b/c it could be aliased, imported, static imported. And that's just a method call.

Future Groovy needs to seamlessly give the user the hook to the local code/AST more like what Lisp does.

Better Splice Options in 2.0?

```
assert x is null // use code as assertion msg
```

```
[meta] def assert(condition as Expression):  
  return [  
    if not $condition:  
      raise AssertionError($condition.ToCodeString())  
  ]
```

Boo meta-methods show a technique also present in Lisp and Template Haskell: How to splice source code into a macro using the \$ operator and how to splice AST back into source code using the [| |] operator (called quasi-quote or Oxford Brackets)

This is currently quite difficult in Groovy. Pulling the source AST into your transformer isn't easy from a visitor because you don't have access to accumulated tree. Pushing AST into source isn't easy because the visitor is all based on side effects and not a functional tree walker... (see next slide)

Missing True “Transformations”

```
void visitMethodCallExpression(MethodCallExpression e)
{
    // given a void return type... how will you transform
    // MethodCallExpression into DeclarationExpression
}
```

It is much easier to route a method call to another different method call than rewrite the code completely. This leads to hacks.

What's needed is a functional style visitor in which all visit methods take an immutable `ASTNode` subtype and return an `ASTNode` value.

You know, you could probably write this as a library on top of Groovy 1.6

Rigid Groovy/Java Syntax

What transform converts this:

```
(let x (+ 5 1))
```

into this:

```
def x = 5 + 1
```

I don't think this is possible. The Groovy syntax is fixed. You can “hijack” a goto label but you can't replace the syntax.

There is a difference between syntax and semantics. AST Transforms allow you to change semantics only.

Maybe now you can appreciate a simple, minimal syntax with all other constructs built as libraries on top of that?

Variable Capture

```
@WithParameterLogging
...
astFactory.build( "def result = new StringBuilder();" )
...
```

```
@WithParameterLogging
def myMethod(String key, String value) {
...
def result = [ ]
...
}
```

Imagine a transform writing a variable declaration into the AST and then a method writing the same variable. This will produce a compile error.

Groovy has “unhygienic” macros (as well as Lisp, right?)

Groovy is missing a gensym operator, which results in naming conflicts. There is not a good solution to this problem and it will get worse over time.

There are 2 types of capture. I'm not clear on details: Argument Capture and Free Symbol Capture. In one, the calling context has something overridden, in other the calling context shadows something.

Compile Time Metaprogramming

Good Things

When nothing else will do

To call functions without evaluating arguments

To modify variables in calling scope

Bad Things

Difficult to write?

Source code clarity

Runtime clarity

“The first step in translating a function into a macro is to ask yourself if you really need to do it. Couldn’t you just as well declare the function inline.”

“When do macros bring advantages? ... Usually the question is not one of advantage, but necessity.”

Both quotes Paul Graham – On Lisp

When is it necessity

1 - in a function call, all the arguments are evaluated before the function is even invoked. You can work around this in a macro.

2 - A macro can generate an expansion containing a variable whose binding comes from the context of the macro call:

```
(defmacro foo (x)
  '(+ ,x y))
```

Compile time or run-time?

Compile Time

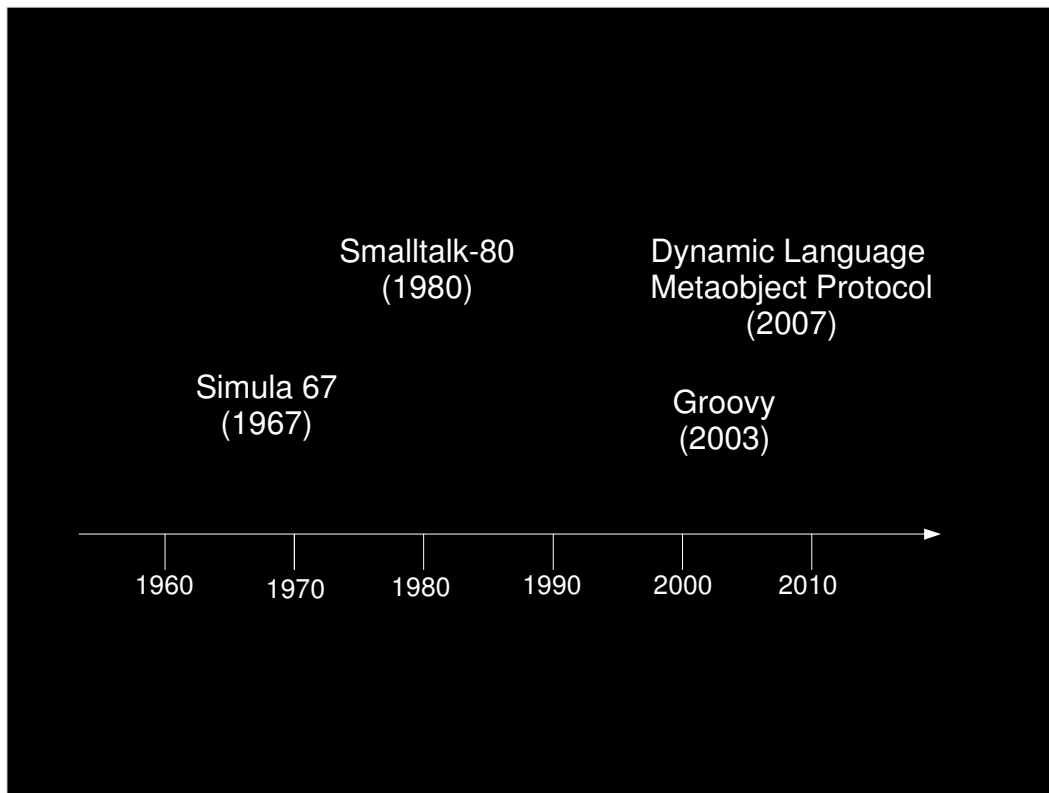
- * Visible in bytecode
- * Supports semantic language changes
- * Low level source access

Runtime

- 1) Simpler
- 2) Better documented
- 3) Easier to debug
- 4) Supports principle of least surprise better

There is a new idea on the mailing list every week. Why should they be added to the language? Why was @Singleton for that matter?

Groovy blurs boundaries between compile & runtime because Groovy is most often compiled seconds before execution.



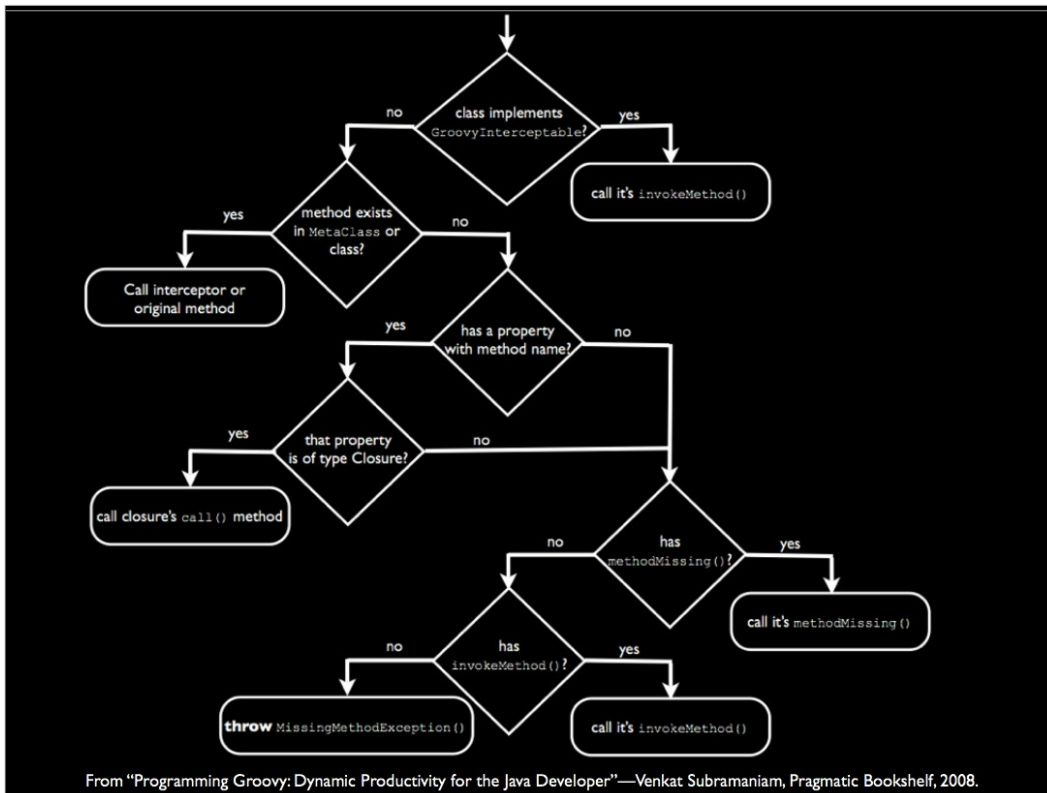
Simula introduced inheritance

Smalltalk introduced doesNotUnderstand:

Groovy introduced invokeMethod

Atila Szegedi introduced JVM Dynamic Language

Metaobject protocol



An adapter based approach to language interoperability would have to replicate this in order to call into Groovy.

JVM Dynamic Language Metaobject Protocol

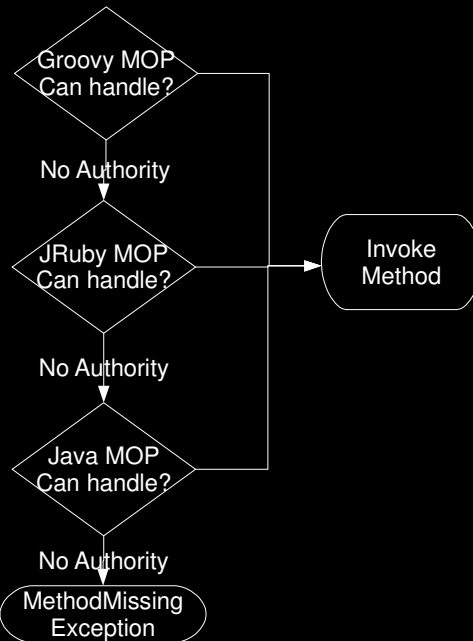
```
((Scriptable)obj).get("foo");
```

vs.

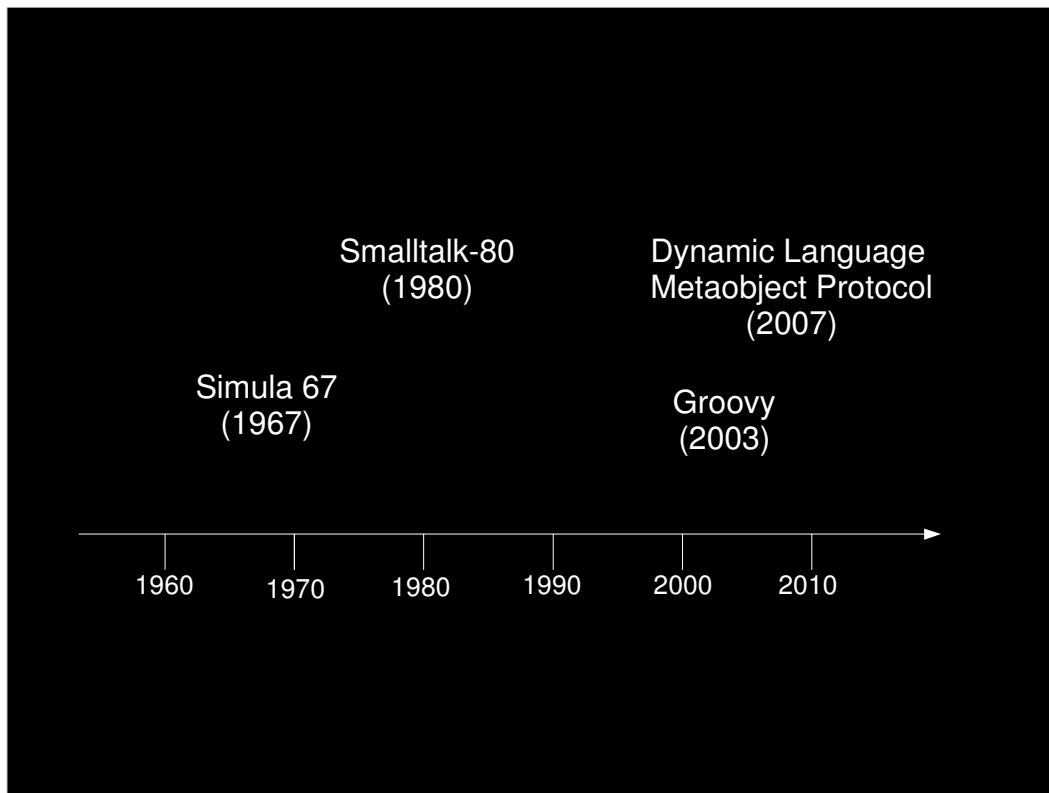
```
metaobjectProtocol.get(obj, "foo");
```

Dynamic MOP provides “navigator” based service layer to locate and invoke methods.

Composability



Key is Composability



The history of programming languages doesn't seem to be about making the right fundamental decisions and having further work build on it. It seems to be about finding ways to safely put more and more flexibility in the hands of the programmers. I relish the idea of one day being able to safely modify method invocation in a language, and the Dynamic MOP seems like a good step forward.

There's a lot we didn't talk about

Actually, not too much missed for compile time.

Get Involved with the prototype. Fix some of the broken unit tests.

Next enhancement should be easier insertion points.

Consider looking at:

Spock Framework, CodeNarc, groovy-core annotations
My blog. GEP-2. Mailing list. What else?

Thank You!

Hamlet D'Arcy

Twitter: HamletDRC

Email: HamletDRC@gmail.com

Web: <http://hamletdarcy.blogspot.com>