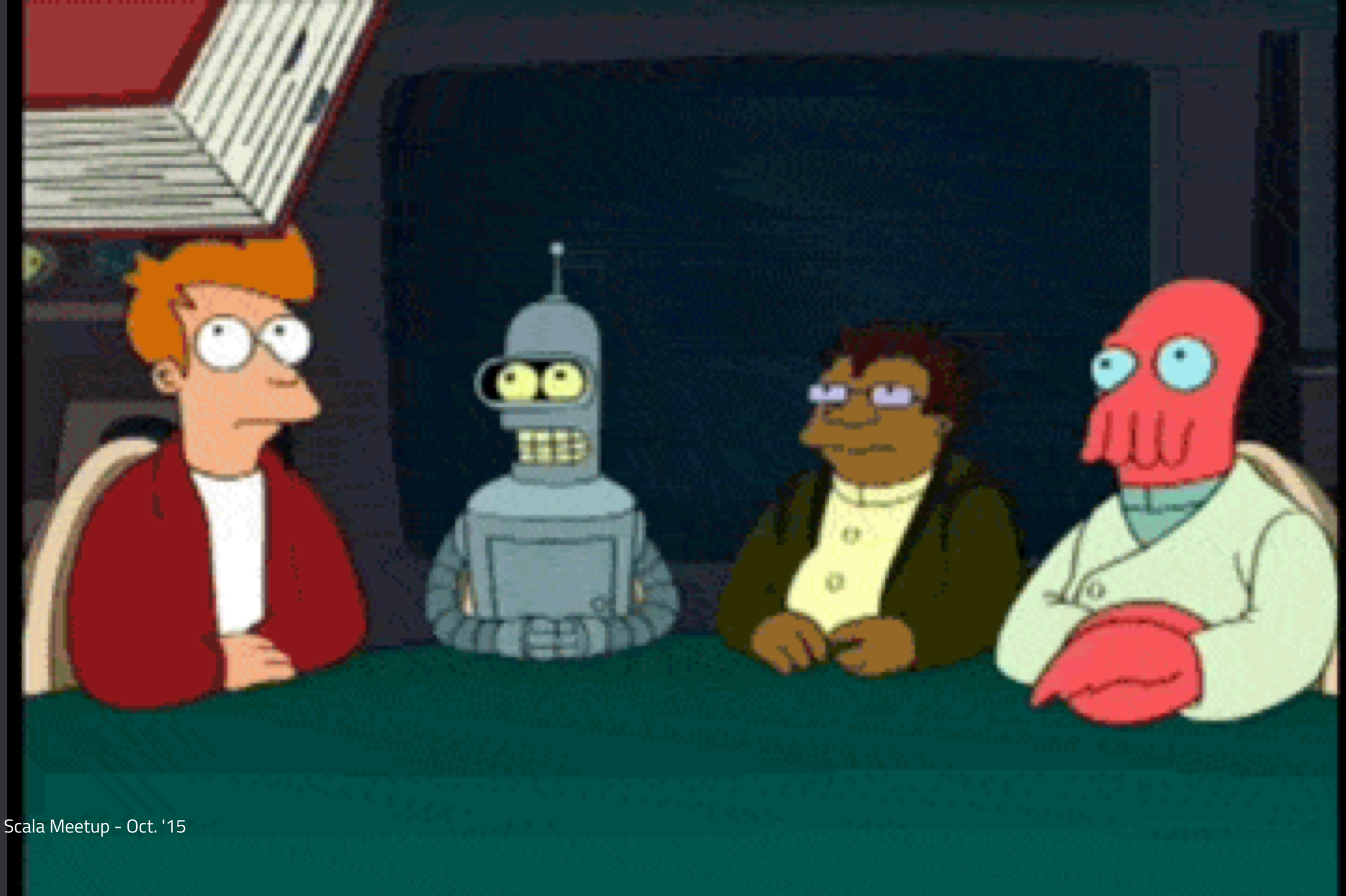
Scala Macros,

or How I Learned To Stop Worrying and Mumble "WTF?!?!"

Brendan McAdams – brendan@boldradius.com

@rit

What Are Macros?



“metaprogramming”

But Seriously, What *Are* Macros?

- 'metaprogramming', from the Latin: 'WTF?'. I mean, "code that writes code".
- Write 'extensions' to Scala which expand out to more complicated code when used. Evaluated at compile time.
- Facility for us to write syntax that feels "built in" to Scala, e.g. String Interpolation:

```
s"Foo: $foo Bar: $bar FooBar: ${foo + bar}"
```

- Annotations that rewrite / expand code:

```
@hello
object Test extends App {
  println(this.hello)
}
```

- ... And a lot more.

Disclaimer: I'm Barely Qualifed To Speak About Macros

- I'm fairly new to Macros – there is a ton to absorb and some of it feels like deep, deep, black magic.
- On the other hand, so *many* Macros talks are given by Deeply Scary Sorcerers and Demigods... who sometimes forget how hard this stuff can be to learn.
- Let's take a look at this through *really fresh*, profusely bleeding, eyeballs.

Once Upon A Time...

- We could pull off some (most?) of what we can do with Macros, by writing compiler plugins.
- Esoteric, harder to ship (i.e. user must include a compiler plugin), not a lot of docs or examples.
- Required *deep* knowledge of the AST: Essentially generating new Scala by hand-coding ASTs.†
- I've done a little bit of compiler plugin work: the AST can be tough to deal with.

† Abstract Syntax Tree. A simple “tree” of case-class like objects to be converted to bytecode... or JavaScript.

A Light Taste of the AST

Given a small piece of Scala code, what might the AST look like?

```
class StringInterp {
  val int = 42
  val dbl = Math.PI
  val str = "My hovercraft is full of eels"

  println(s"String: $str Double: $dbl Int: $int Int Expr: ${int * 1.0}")
}
```

My God... It's Full of ... Uhm

```
Block(
  List(
    ClassDef(Modifiers(), TypeName("StringInterp"), List(), Template(
      List(Ident(TypeName("AnyRef"))), noSelfType, List(DefDef(Modifiers(), termNames.CONSTRUCTOR,
        List(),
        List(List()),
        TypeTree(), Block(List(Apply(Select(Super(This(typeNames.EMPTY), typeNames.EMPTY),
        termNames.CONSTRUCTOR), List())), Literal(Constant(())))), ValDef(Modifiers(), TermName("int"),
        TypeTree(), Literal(Constant(42))), ValDef(Modifiers(), TermName("dbl"), TypeTree(),
        Literal(Constant(3.141592653589793))), ValDef(Modifiers(), TermName("str"), TypeTree(),
        Literal(Constant("My hovercraft is full of eels"))), Apply(Select(Ident(scala.Predef),
        TermName("println")), List(Apply(Select(Apply(Select(Ident(scala.StringContext), TermName("apply")),
        List(Literal(Constant("String: ")), Literal(Constant(" Double: ")), Literal(Constant(" Int: ")),
          Literal(Constant(" Int Expr: ")), Literal(Constant("")))), TermName("s")),
        List(Select(This(TypeName("StringInterp")), TermName("str")), Select(This(TypeName("StringInterp")),
          TermName("dbl")), Select(This(TypeName("StringInterp")), TermName("int")),
          Apply(Select(Select(This(TypeName("StringInterp")), TermName("int")), TermName("$times")),
            List(Literal(Constant(1.0)))))))))
    ))), Literal(Constant(())))
```

Enter The Macro

- Since Scala 2.10, Macros have shipped as an experimental feature in Scala 2.10.
- Seem to have been adopted fairly quickly: I see them all over the place.
- By example, more than a few SQL Libraries have added **`sql`** string interpolation prefixes which generate proper JDBC Queries.
- AST Knowledge can be somewhat avoided, with some really cool tools to generate it for you.
- Much easier than compiler plugins, to add real enhanced functionality to your projects.


HAIL SCIENCE!

Macros, The AST, and Def Macros

- Macros are still really built with the AST, but lately Macros provide tools to generate ASTs from code (which is what I use, mostly).
- The first, and simplest, is **`reify`**, which we can use to generate an AST for us.
- Let's look first at ‘Def’ Macros, which let us write Macro methods.[‡]

[‡] I've stolen some code from the official Macros guide for this.

A Def Macro for printf

First, we need to define a **`printf`** method which will ‘proxy’ our Macro definition:

```
// Import needed if you're *writing* a macro
import scala.language.experimental.macros
def printf(format: String, params: Any*): Unit = macro printf_impl
```

This is our macro *definition*. We also need an *implementation*.

A Def Macro for printf

```
import scala.reflect.macros.Context
def printf_impl(c: Context)(format: c.Expr[String],
                            params: c.Expr[Any]*): c.Expr[Unit] = ???
```

We'll also want to import (in our **`printf_impl`** body) **`c.universe._`**.

This provides a lot of routine types & functions (such as **`reify`**).

Generating The Code for `printf`

Here's our first problem: when **`printf`** calls **`printf_impl`** the Macro implementation converts all of our *values* into *syntax trees*. But we can use the AST case classes to extract:

```
val Literal(Constant(s_format: String)) = format.tree
```

Generating The Code for `printf`

We then need code to split out the format string, and substitute parameters:

```
val paramsStack = Stack[Tree]((params map (_.tree)): _*)
val refs = s_format.split("(?<=%[\\w%])|(?=%[\\w%])") map {
  case "%d" => precompute(paramsStack.pop, typeOf[Int])
  case "%s" => precompute(paramsStack.pop, typeOf[String])
  case "%%" => Literal(Constant("%"))
  case part => Literal(Constant(part))
}
```

You'll note some references to **`precompute`**... which is another fun ball full of AST.

Generating The Code for `printf`

`precompute` (a function we write ourselves) helps us convert our varargs **`params`** into AST statements we can reuse:

```
val evals = ListBuffer[ValDef]()
def precompute(value: Tree, tpe: Type): Ident = {
  val freshName = TermName(c.fresh("eval$"))
  evals += ValDef(Modifiers(), freshName, TypeTree(tpe), value)
  Ident(freshName)
}
```

In particular, we're generating a substitute name, and saving into **`evals`** all of the params into value definitions.

Generating The Code for `printf`

Lastly, we stick it all together. Here, **`reify`** is used to simplify the need to generate AST objects, doing it *for* us:

```
val stats = evals ++ refs.map { ref =>
  reify( print(c.Expr[Any](ref).splice) ).tree
}
// our return from `printf_impl`
c.Expr[Unit](Block(stats.toList, Literal(Constant(()))))
```

Note we're using **`print`**, not **`println`**, so each individual **`ref`** (a.k.a block of string) is printed, using a value from **`evals`**. **`splice`** helps us graft a **`reify`** block onto the syntax tree.

Using `printf`

It works pretty much as you'd expect:*

```
scala> printf("%s: %d", "The Answer", 42)
The Answer: 42
```

We could, on the console, use **`reify`** to see how Scala expands our code:

```
import scala.reflect.runtime.universe._
reify(printf("%s: %d", "The Answer", 42))

res1: reflect.runtime.universe.Expr[Unit] =
  Expr[Unit](PrintfMacros.printf("%s: %d", "The Answer", 42))
```

---

* NOTE: You need to define your macros in a *separate* project / library from anywhere you call it.

Peeking at AST Examples for “Inspiration”

Remember my first example of the AST? I actually printed it out using **`reify`**:

```
println(showRaw(reify {
  class StringInterp {
    val int = 42
    val dbl = Math.PI
    val str = "My hovercraft is full of eels"

    println(s"String: $str Double: $dbl Int: $int Int Expr: ${int * 1.0}")
  }
}.tree))
```

`.tree` will replace the **`reify`** ‘expansion’ code with the AST associated. **`showRaw`** converts it to a printable format for us.


COMEDY
CENTRAL

quasiquotes for More Sanity

- There's really no way – yet – to avoid the AST Completely. But the Macro system continues to improve to give us ways to use it less and less.
- quasiquotes, added in Scala 2.11, lets us write the equivalent of String Interpolation code that ‘evals’ to a Syntax Tree.
- We aren't going to go through a Macro build with quasiquotes (yet), but let's look at what they do in the console...

quasiquotes in Action

Setting Up Our Imports

There are some implicits we need in scope for QuasiQuotes Ah, the joy of imports...

```
import language.experimental.macros
import reflect.macros.Context
import scala.annotation.StaticAnnotation
import scala.reflect.runtime.{universe => ru}
import ru._
```

Now we're ready to generate some Syntax Trees!

quasiquotes in Action

Writing Some Trees

quasiquotes look like String Interpolation, but we place a **q** in front of our string instead of **s**:

```
scala> q"def echo(str: String): String = str"


res4: reflect.runtime.universe.DefDef =
    def echo(str: String): String = str
```

quasiquotes in Action

Writing Some Trees

```
scala> val wtfException = q"case class OMGWTFBBQ(message: String = null) extends Exception with scala.util.control.NoStackTrace"

wtfException: reflect.runtime.universe.ClassDef =
case class OMGWTFBBQ extends Exception
    with scala.util.control.NoStackTrace
    with scala.Product
    with scala.Serializable {
  <caseaccessor> <paramaccessor> val message: String = _;
  def <init>(message: String = null) = {
    super.<init>();
    ()
}
```

Extracting with quasiquotes

It turns out, quasiquotes can do extraction too, which I find sort of fun.

```
scala> val q"case class $cname(..$params) extends $parent with ..$traits { ..$body }" = wtfException
cname: reflect.runtime.universe.TypeName = OMGWTFBBQ
params: List[reflect.runtime.universe.ValDef] = List(<caseaccessor> <paramaccessor> val message: String = null)
parent: reflect.runtime.universe.Tree = Exception
traits: List[reflect.runtime.universe.Tree] = List(scala.util.control.NoStackTrace)
body: List[reflect.runtime.universe.Tree] = List()
```

Macro Paradise

- It is worth mentioning that the Macro project for Scala is evolving *quickly*.
- They release and add new features *far more frequently* than Scala does.
- "Macro Paradise" is a compiler plugin meant to bring the Macro improvements into Scala[¶] as they become available.
- One of the features currently in Macro Paradise is Macro Annotations.
- You can learn more about Macro Paradise at http://docs.scala-lang.org/overviews/macros/paradise.html

[¶] focused on reliability with the current production release of Scala

Macro Annotations

- Macro Annotations are designed to let us build annotations that expand via Macros.
- The possibilites are endless, but there's a great demo repository that shows how to combine Annotations & quasiquotes to rewrite a class...
- You can find this code at https://github.com/scalamacros/sbt-example-paradise

Macro Annotations

A Very Basic Demo

At the beginning of the talk, I showed an odd piece of code. It was annotated, but also referenced a variable that didn't exist:

```
@hello
object Test extends App {
  println(this.hello)
}
```

`hello` doesn't appear to be a member of **`Test`**. It certainly *isn't* defined in **`App`**.

Instead, the **`@hello`** annotation represents a Macro, which rewrites our object definition.

Macro Annotations

The `hello` Macro

```
import scala.reflect.macros.Context
import scala.language.experimental.macros
import scala.annotation.StaticAnnotation

class hello extends StaticAnnotation {
  def macroTransform(annottees: Any*) = macro helloMacro.impl
}
```

The annotation is declared; Let's look at the implementation.

Macro Annotations

The `hello` Macro

```
object helloMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._
    import Flag._
    val result = ??? // we'll look at this in a second
    c.Expr[Any](result)
  }
}
```

All we need now is the code for **`result`** to generate an AST.

Macro Annotations

The `hello` Macro

Finally, the value of result is generated via quasiquotes, rewriting the annotated object:

```
val result = {
  annottees.map(_.tree).toList match {
    case q"object $name extends ..$parents { ..$body }" :: Nil =>
      q"""
        object $name extends ..$parents {
          def hello: ${typeOf[String]} = "hello"
          ..$body
        }
      """
  }
}
```


BRILLIANT!

Closing Thoughts

Macros are undoubtedly cool, and rapidly evolving. But be cautious.

“When all you have is a hammer, everything starts to look like a thumb...”
— me

Macros can enable great development, but also hinder it if overused. Think carefully about their introduction, and their impact on your codebase.

Questions?