Advanced Concepts in  Scala

Running Things Before Runtime

Nima Taheri

Jan 2017

Table of Contents

- Why Scala ?
- Dive into Scala compiler
- Blackbox, Whitebox & Annotation macros
- Meta-programming using Scala.meta

Table of Contents

Why Scala ?

- Modern Design
- Clean & Concise
- Type Safe
- Easily Scales
- Feature-Rich
- Open-Source, Academic, Community-Driven

Why Scala ? - Modern Design

- Pure Object-Oriented
  - No static field
  - Primitives are objects
- Traits
  - No need for abstract class
  - Great tool for modularity
- Functional
  - Higher order functions
  - Currying
  - Nested function

Why Scala ? - Clean & Concise

- Type Inference
- Pattern Matching
- Closure
- Placeholder Syntax
- Rich Immutable Collections
- Macro
- Lazy
- Avoid Null
- No Loops
- No Unsafe Casting

Why Scala ? - Type Safe

- Strong Type System
- Promotes Compile-Time Meta-Programming
- Type-Level Libraries
- Pattern Matching instead of Type Casting

Why Scala ? - Easily Scales

- Immutable Objects
- Future, Execution Context
- Lightweight Actor-system (Akka)
- Modularity Facilities

Table of Contents

- Dive into Scala compiler

Dive into Scala Compiler

source.scala

files*.class

Dive into Scala Compiler

- Overview
- Phases
- Internal Data Structures
- Inspection and Creation of AST
- Add a new phase
- A simple plugin

Dive into Scala Compiler - Overview

- Internal Data Structures

Tree

Symbol

Type

Universe

- Control Flow

24 phases

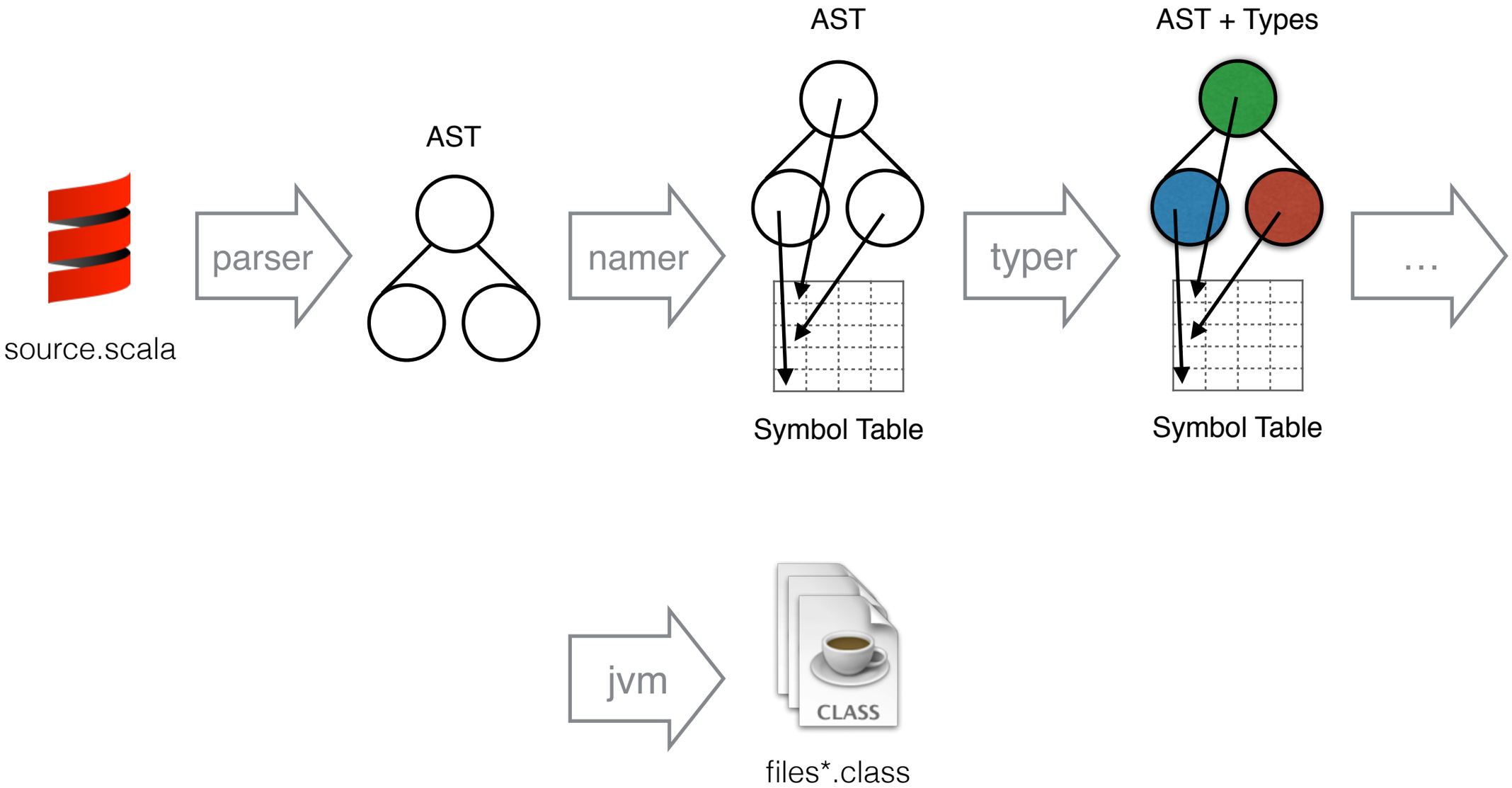
Sequentially applied

```
class Global {
  class Run {
    def compile(filenames: List[String]) {
      // produce binaries out of scala sources
    }
  }
}

trait Phase { def apply(t:Tree):Tree }

val phases: List[Phase] = ___

object parser { def apply(code: String):Tree }

def compile(code: String): Tree =
    phases.foldLeft(parser(code))(_ apply _)
```

Dive into Scala Compiler - Phases (1)

```
$ scalac -Xshow-phases

    phase name  id  description
    ----------  --  -----------
        parser   1  parse source into ASTs, perform simple desugaring
         namer   2  resolve names, attach symbols to named trees
packageobjects   3  load package objects
         typer   4  the meat and potatoes: type the trees
        patmat   5  translate match expressions
superaccessors   6  add super accessors in traits and nested classes
    extmethods   7  add extension methods for inline classes
       pickler   8  serialize symbol tables
     refchecks   9  reference/override checking, translate nested objects
       uncurry  10  uncurry, translate function values to anonymous classes
        fields  11  synthesize accessors and fields, add bitmaps for lazy vals
     tailcalls  12  replace tail calls by jumps
    specialize  13  @specialized-driven class and method specialization
explicitouter  14  this refs to outer pointers
      erasure  15  erase types, add interfaces for traits
  posterasure  16  clean up erased inline classes
    lambdalift  17  move nested functions to top level
  constructors  18  move field definitions into constructors
      flatten  19  eliminate inner classes
        mixin  20  mixin composition
      cleanup  21  platform-specific cleanups, generate reflective calls
   delambdafy  22  remove lambdas
          jvm  23  generate JVM bytecode
     terminal  24  the last phase during a compilation run
```

Dive into Scala Compiler - Phases (2)



Dive into Scala Compiler - Phases (3)

```scala
object SimpleClass {

  def join(str1: String)(str2: String) = {
    s"$str1 + $str2"
  }

}
```

TYPER

```scala
object SimpleClass {

  def join(str1: String)(str2: String): String = {
    s"$str1 + $str2"
  }

}
```

Dive into Scala Compiler - Phases (3)

```scala
object SimpleClass {

  def join(str1: String)(str2: String): String = {
    s"$str1 + $str2"
  }

}
```


```scala
object SimpleClass {

  def join(str1: String, str2: String): String = {
    s"$str1 + $str2"
  }

}
```

Dive into Scala Compiler - Phases (4)

```
$ scalac -Xprint:packageobjects SimpleClass.scala

package <empty> {
  object SimpleClass extends scala.AnyRef {
    def <init>() = {
      super.<init>();
      ()
    };
    def join(str1: String)(str2: String) =
        StringContext("", " + ", "").s(str1, str2);
    def main(args: Array[String]) =
        println(join("Nima")("Taheri"))
  }
}
```

TYPER

```
$ scalac -Xprint:typer SimpleClass.scala

package <empty> {
  object SimpleClass extends scala.AnyRef {
    def <init>(): SimpleClass.type = {
      SimpleClass.super.<init>();
      ()
    };
    def join(str1: String)(str2: String): String =
      scala.StringContext.apply("", " + ", "").s(str1, str2);
    def main(args: Array[String]): Unit =
      scala.Predef.println(SimpleClass.this.join(“Nima”)("Taheri"))
  }
}
```

Dive into Scala Compiler - Data Structures - Trees

```
$ scalac -Xprint:typer
         -Yshow-trees SimpleClass.scala
```

```
object Main {
  def main(args: Array[String]) {
    val name = "--> Nima"
    println(name.substring(4))
  }
}
```

SimpleClass.scala

```
$ scalac -Xprint:typer SimpleClass.scala
```

```
package <empty> {
  object Main extends scala.AnyRef {
    def <init>(): Main.type = {
      Main.super.<init>();
      ()
    };
    def main(args: Array[String]): Unit = {
      val name: String = "--> Nima";
      scala.Predef.println(name.substring(4))
    }
  }
}
```

```
PackageDef(
  "<empty>"
  ModuleDef( // object Main
    <module>
    "Main"
    Template(
      "scala"."AnyRef" // parents
      ValDef(
        private
        "_"
        <tpt>
        <empty>
      )
      // 2 statements
      DefDef(
        0
        "<init>"
        []
        List(Nil)
        <tpt>
        Block(
          Apply(
            super."<init>"
            Nil
          )
          ()
        )
      )
      DefDef(
        0
        "main"
        []
        // 1 parameter list
        ValDef(
          <param>
          "args"
          AppliedTypeTree(
            "Array"
            "String"
          )
          <empty>
        )
        "scala"."Unit"
        Block(
          ValDef(
            0
            "name"
            <tpt>
            "--> Nima"
          )
          Apply(
            "println"
            Apply(
              "name"."substring"
              4
            )
          )
        )
      )
    )
  )
)
```

PackageDef, ModuleDef, DefDef, Block, Apply <: **Tree**

Dive into Scala Compiler - Data Structures (Tree / Symbol / Type)

- Trees are the foundation for Scala's AST

- **Parser** phase creates Trees, which are immutable except for Position, Symbol, Type

- **Namer** phase assigns Symbols to Trees

  A symbol contains all attributes of a tree
  (e.g. symbol of a class-tree has all information regarding its fields, methods, etc.)

- **Typer** phase assigns Type to Trees

- A universe is always required

  Use import scala.reflect.runtime.universe._ for playing in REPL/Runtime

Dive into Scala Compiler - Inspecting AST

- Pattern matching manually (Using ModuleDef, DefDef, Block, Apply)

```
tree match {
  case ModuleDef(_, "module-name", body) => …
}
```

- Pattern matching using quasiquotes

```
tree match {
  case q"$mods object $tname {..$body }" => …
}
```

- Tree Traverser

```
class MyTraverser extends Traverser {
  override def traverse(t: Tree): Unit = {
    val tree = super.traverse(t)
      tree match {
        case q"$expr(...$exprss)" => …
      }
  }
}
```

Dive into Scala Compiler - Creating AST

- Manually (Using ModuleDef, DefDef, Block, Apply)

- Using reify { }

```
reify { println(“nima”) }
Apply(Select(Ident(scala.Predef), TermName("println")), List(Literal(Constant("nima"))))
```

- Using quasiquotes

```
q”””  println(“nima”)  ”””
Ident(TermName("println")), List(Literal(Constant(“nima")))
```

- Tree Transformer

```
class MyTransformer extends Transformer {
  override def transform(t: Tree): Tree = {
    val tree = super.traverse(t)
      tree match {
        case q"$expr(...$exprss)" => q”???”
      }
  }
}
```

Dive into Scala Compiler - Add a new phase (1)

- Adding a new phase to scale-compiler source
- Writing a compiler plugin

extends universe

```
Class MyPlugin(val global: Global) extends Plugin {
  override val name: String = "NAME"
  override val description: String = "DESCRIPTION"
  override val components = List[PluginComponent](MyPluginComponent)

  object MyPluginComponent extends PluginComponent {
    override val runsAfter = List("parser")
    import global._

    def newPhase(prev: Phase): StdPhase = new StdPhase(prev) {
      def apply(unit: CompilationUnit) {
        val tree = unit.body
          ...
      }
    }
  }
}
```

Dive into Scala Compiler - Add a new phase (2)

- Add plugin to scalar

```
$ scalac -Ylog:{NAME} -Xplugin:{plugin-jar-or-classpath}
```

- Check if plugin is loaded

```
$ scalac -Xplugin:{plugin-jar-or-classpath} -Xplugin-list
```

- Check if new phase is added

```
$ scalac -Xplugin:{plugin-jar-or-classpath} -Xshow-phases

  phase name  id  description
    ----------  --  -----------
        parser   1  parse source into ASTs, perform simple desugaring
          NAME   2  DESCRIPTION
         namer   3  resolve names, attach symbols to named trees
packageobjects   4  load package objects
         typer   5  the meat and potatoes: type the trees
```

Dive into Scala Compiler - A simple plugin (1)

```
def newPhase(prev: Phase): StdPhase = new StdPhase(prev) {
  def apply(unit: CompilationUnit) {
    new MyTransformer().transformUnit(unit)
  }
}

class MyTransformer extends Transformer {
  override def transform(t: Tree): Tree = {
    val tree = super.transform(t)
    tree match {
      case node @ q"$mod val $v: $tpe = $value" if !value.isEmpty =>
        // log the node
        log(s"Writing stdout logger for >> ${show(node)}")
        // change initial-value expression to a closure
        // which prints the value before assigning it to the variable
        q"$mod val $v: $tpe = { val $$tmp = $value; println($$tmp); $$tmp };"
      case _ =>
        tree
    }
  }
}
```

Dive into Scala Compiler - A simple plugin (2)

```
$ scalac -Ylog:{NAME} -Xprint:NAME -Xplugin:{plugin-jar-or-classpath} SimpleClass.scala
```

```scala
object Nima {

  def main(args: Array[String]) {
    val x: Int = 100
    val y: Int = 200
    val z: Int = 300
    println(s"result: $x")
  }

}
```

SimpleClass.scala

console output

```
[log NAME] Writing stdout logger for >> val x: Int = 100
[log NAME] Writing stdout logger for >> val y: Int = 200
[log NAME] Writing stdout logger for >> val z: Int = 300

 object Nima {
    def main(args: Array[String]) {
      val x: Int = {
        val $tmp = 100
        println($tmp)
        $tmp
      }
      val y: Int = {
        val $tmp = 200
        println($tmp)
        $tmp
      }
      val z: Int = {
        val $tmp = 300
        println($tmp)
        $tmp
      }
    println(v1 + v2)
  }
}
```

Table of Contents

Blackbox, Whitebox & Annotation macros

- Overview
- Cyclic dependency problem
- Type erasure problem
- Quasiquotes
- Write a macro

Blackbox, Whitebox & Annotation macros - Overview (1)

- Macros
  Compiler passes a Context to macros that contains a universe

- Blackbox / Whitebox macros
  Work like a method
  Receive expression trees, produce expression trees
  Whitebox macros: result-type in method signature can be dynamic

- Annotation macros
  Require macro-paradise plugin to work
  Can manipulate an annotated structure (class, object, method, etc.)

Blackbox, Whitebox & Annotation macros - Overview (2)

- Why macros ?

  Code generation (eliminating boilerplates)

  Program Verification

  Refactoring

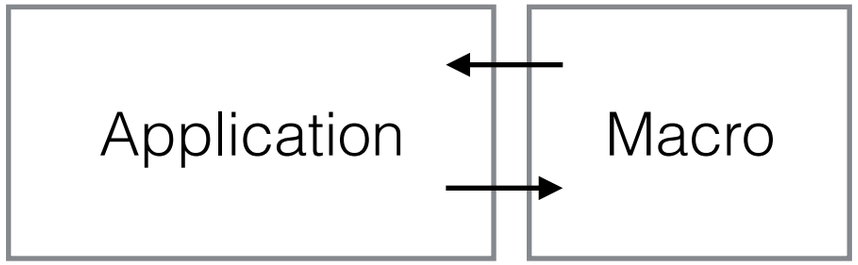
  etc.

- Why compile-time macros ?

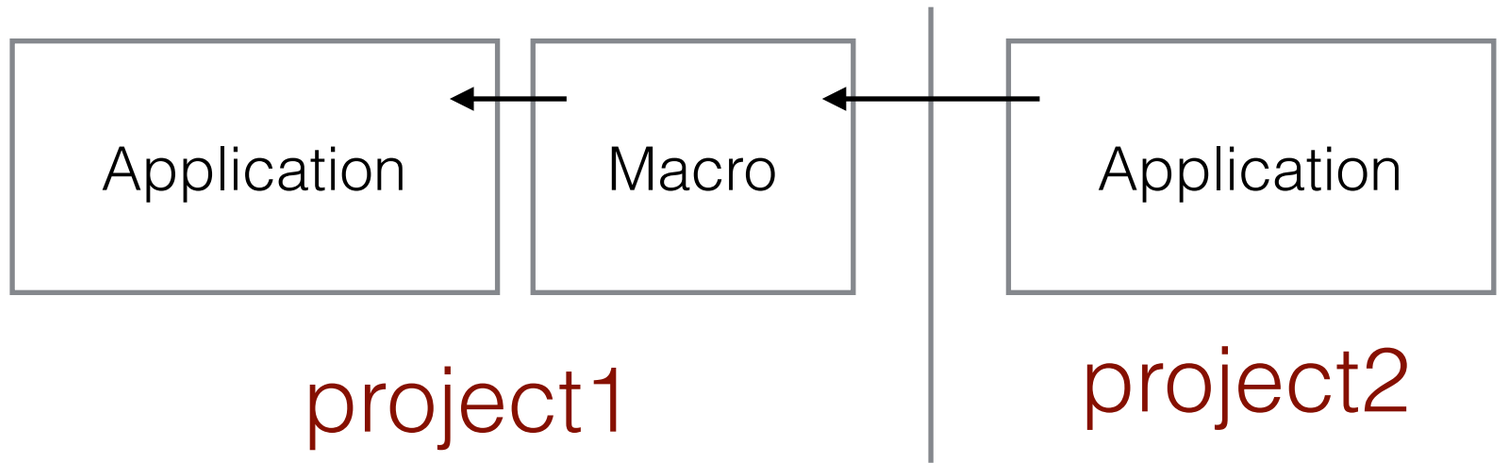
  Type Safety

  Higher Runtime Performance

Blackbox, Whitebox & Annotation macros - Cyclic Dependency Problem



This could go forever !



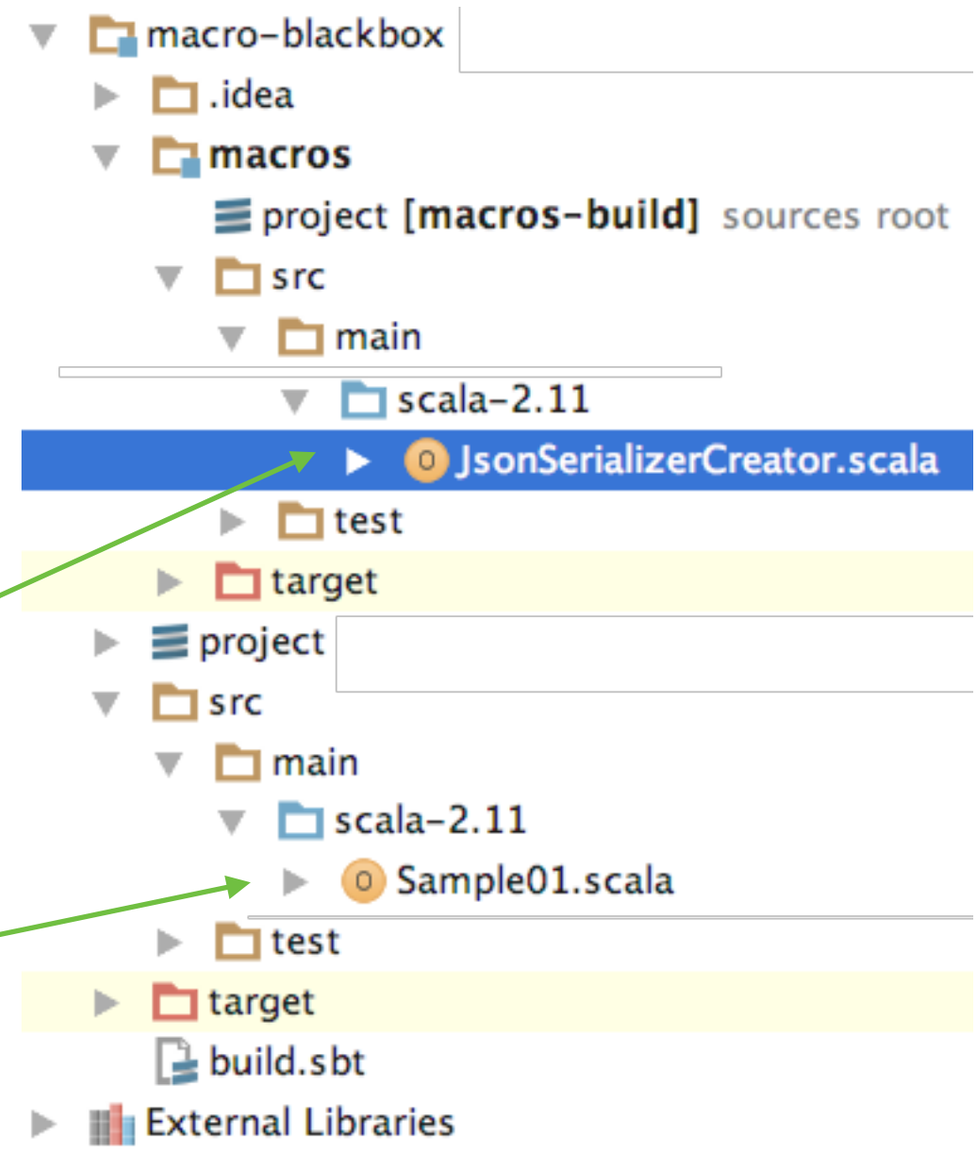
Blackbox, Whitebox & Annotation macros - Cyclic Dependency Problem

```
lazy val macros =
  project in file("macros"))
    .settings(Seq(
    libraryDependencies +=
      "org.scala-lang" %
      "scala-reflect" %
      "2.11",
    )

dependsOn(macros)
```

build.sbt

Macros

Application



Blackbox, Whitebox & Annotation macros - Type Erasure Problem

```scala
def four[T](t: T): List[T] = {
  t :: t :: t :: Nil
}
```


```scala
def four(t: Object): List = {
  t :: t :: t :: Nil
}
```

- Java does not keep generic information in runtime
- Solution is TypeTags

  Scala compiler generates and injects it for you

  As an implicit object of TypeTag[T] which contains all type-info

```scala
def four[T: TypeTag](t: T)($evidence: TypeTag[T]): List[T] = {
  val extractedType = $evidence.tpe
  t :: t :: t :: Nil
}
```

Blackbox, Whitebox & Annotation macros - Type Erasure Problem

- For types with unknown type-argument:
  use WeakTypeTags
  (e.g. List[T])

```
def weakParamInfo[T](x: T)(implicit tag: WeakTypeTag[T]) = {
  val targs = tag.tpe match { case TypeRef(_, _, args) => args }
  println(s"type of $x has type arguments $targs")
}
def foo[T] = weakParamInfo(List[T]())
// foo: [T]=> Unit
```

Blackbox, Whitebox & Annotation macros - Quasiquote

- Generates AST the easy way through macros
- Syntax summary
  http://docs.scala-lang.org/overviews/quasiquotes/syntax-summary.html

Object	`q"$mods object $tname extends { ..$earlydefns } with ..$parents { $self => ..$body }"`	ModuleDef

Expressions	q"import $ref.{..$sels}”
Definitions	q"{ ..$stats }"
Types	tq”$tpname"
Patterns	pq"$name @ $pat”

Blackbox, Whitebox & Annotation macros - Quasiquote

- Can be used for pattern-matching

```
tree match {
  case q"""$mods object $tname extends { ..$earlydefns }
              with ..$parents { $self => ..$body }""") =>
    println(showRaw(body))
}
```

- Can be used for custom AST creation

```
val myField = q""" val $tname = “nima” """
```

- show() gives scale-like representation

  val myField = “nima"

- showRaw() gives tree representation

  ValDef(Modifiers(), TermName(“myField"), TypeTree(), Literal(Constant("nima")))

Blackbox, Whitebox & Annotation macros - Write a macro

- A typical whitebox/blackbox macro

Could be `c.Tree`

```
import scala.language.experimental.macros
import scala.reflect.macros.blackbox/whitebox.Context

def myMethod(v1: Int): Int  = macro macroOfMyMethod
def macroOfMyMethod(c: Context)(v1: c.Expr[Int]): c.Expr[Int] = {
  import c.universe._
  v1
}
```

Should have same parameters

Universe Provider

- With generic type

```
def myMethod[T](v1: T): Int  = macro macroOfMyMethod
def macroOfMyMethod[T: c.weakTypeTag](c: Context)(v1: c.Tree): c.Expr[Int] = {
  import c.universe._
  val tType = c.weakTypeTag[T].tpe
  c.Expr(v1)
}
```

Blackbox, Whitebox & Annotation macros - Write a macro

- Don’t forget to add scala-reflection to your build.sbt

```
libraryDependencies +=
  org.scala-lang" % "scala-reflect" % scalaVersion.value
```

Blackbox, Whitebox & Annotation macros - Sample blackbox macro (1)

```
def createSerializer[T]: T => String = macro createSerializerMacro[T]

def createJsonSerializer[T: c.WeakTypeTag](c: Context): c.Tree = {
  import c.universe._
  val caseClass = weakTypeOf[T].typeSymbol
  val jsonSerializerType = tq"""JsonSerializer[$caseClass]"""
  val fields = caseClass.asClass.primaryConstructor.asMethod.paramLists.head
  def toJsonValue(s: Symbol) = {
    val theVal = s.name.toTermName
      q""" t.$theVal.toString """
  }
  val commaSeparatedFields = fields.map { field =>
    q""" "\"" + ${field.name.decodedName.toString} + "\":" + ${toJsonValue(field)} """
  }
  val jsonSerializer = q"""
     (t: $caseClass) => {
       "{" +
       ${commaSeparatedFields.reduceLeft( (res, t) => q""" $res + ", " + $t """ )} +
       "}"
     }
   """
  jsonSerializer
}
```

Blackbox, Whitebox & Annotation macros - Sample blackbox macro (2)

```scala
object Sample01 {

  def main(args: Array[String]) = {
    val serializer = createSerializer[ SampleCaseClass01 ]
    val anObject = SampleCaseClass01(field01 = 10, field02 = "Nima")
    val output = serializer(anObject)
    println(output)
  }

}

case class SampleCaseClass01(
  field01: Int,
  field02: String
)
```

```
{"field01":10, "field02":Nima}
```

Console Output

Blackbox, Whitebox & Annotation macros - Sample whitebox macro (1)

```scala
import scala.language.experimental.macros
import scala.reflect.macros.whitebox.Context

object WhiteBox {

  def mix: Any = macro mixMacro

  def mixMacro(c: Context): c.Tree = {
    import c.universe._
    q"""
       case class Foo(a: Int, b: Int)
       Foo(1,2)
     """
  }
}
```

Application can see the type macro has created

```scala
def main(args: Array[String]) = {
  val z = mix
  println(s"  a = ${z.a}")
  println(s"  b = ${z.b}")
}
```

Blackbox, Whitebox & Annotation macros - Sample annotation macro (1)

- Add macro-paradise library to compiler plugins

```
resolvers += Resolver.sonatypeRepo("releases")
addCompilerPlugin(
  "org.scalamacros" % "paradise" % "2.1.0" cross CrossVersion.full)
```

- Manipulate AST of the annotated element
  Could be a parameter, method, class, object
  Will be applied to the parent nodes in AST as well (study ref.)

```
@myannotation
object Test extends App {
  @myannotation
  def myMethod: Int = ???
  def myMethod(@myannotation myParam: Int) = ???
}
```

Blackbox, Whitebox & Annotation macros - Sample annotation macro (2)

- A very simple annotation-macro
  add “hello” method to its annotated object

```
object helloMacro {
  def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
    import c.universe._
    val result = {
      annottees.map(_.tree).toList match {
        case Seq(q"$mods object $tname extends { ..$earlydefns } with ..$parents { $self => ..$body }") =>

          val helloMethod = body :+ q""" def hello = "hello" """
          q"$mods object $tname extends { ..$earlydefns } with ..$parents { $self => ..$helloMethod }"
      }
    }
    c.Expr[Any](result)
  }
}

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

Table of Contents

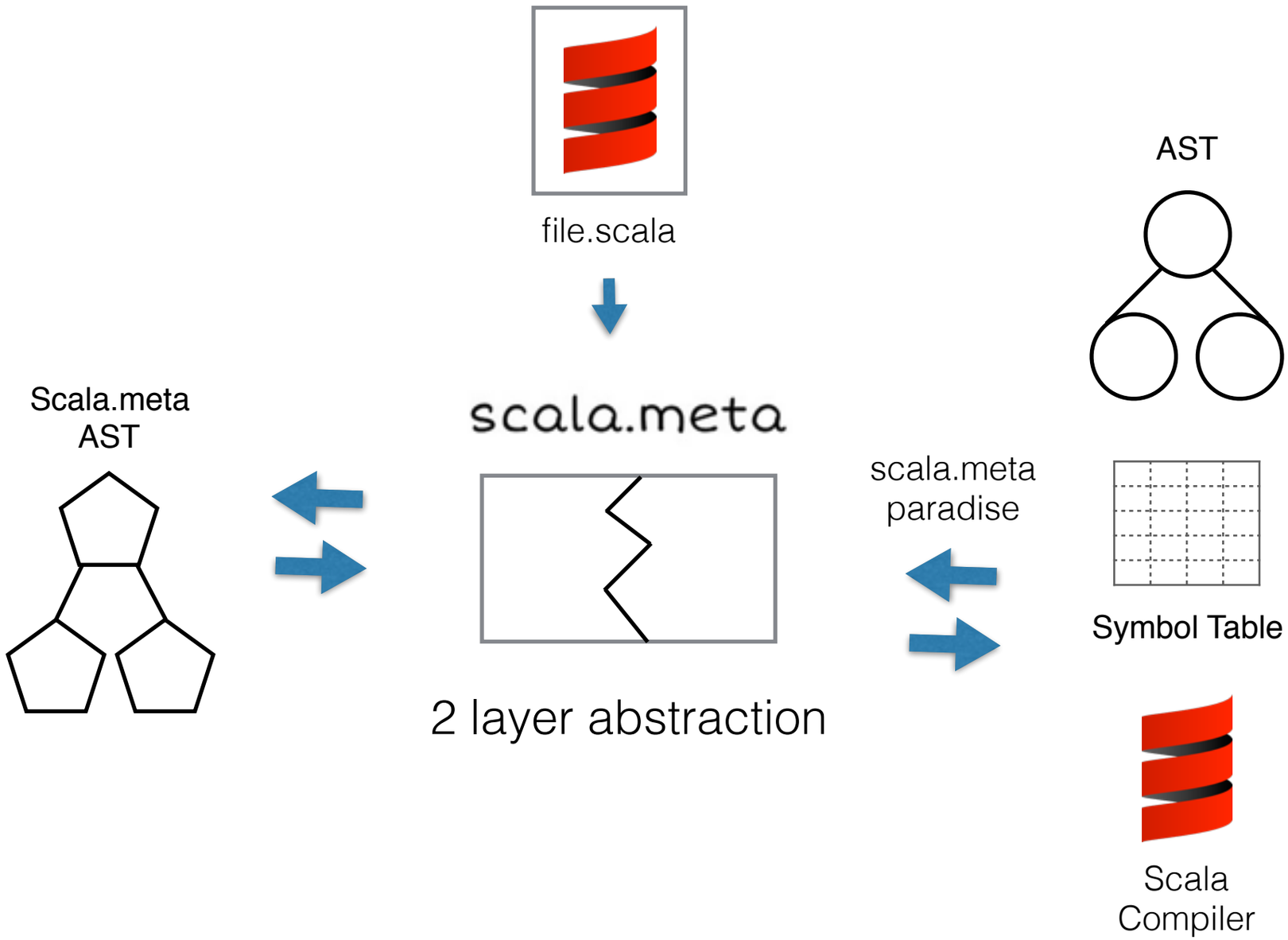
Meta-programming using Scala.meta

- Motivations
- Architecture
- Facilities
- Using Scala.meta

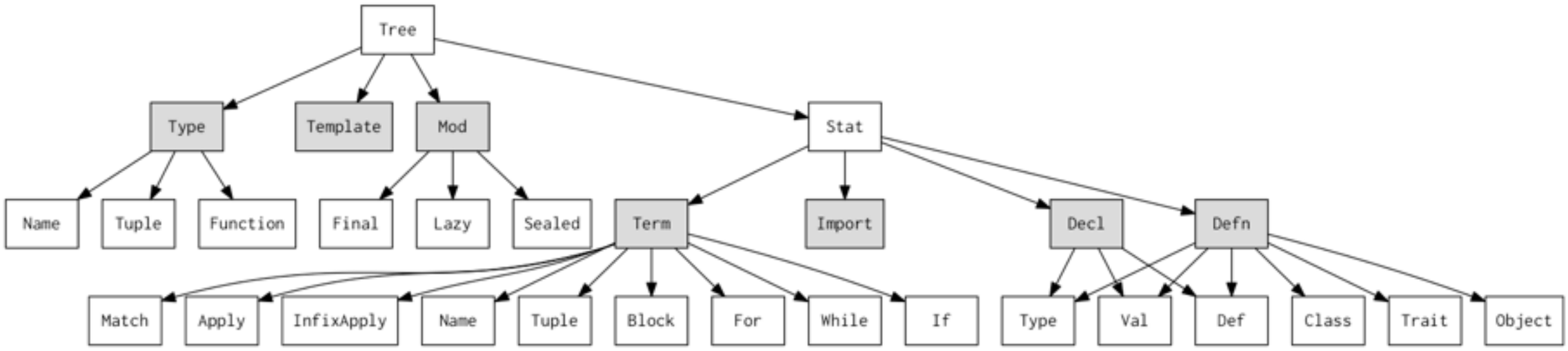
Scala Meta - Motivations

- Scala reflection is difficult to learn
- Compiler internals should not be exposed outside
  - Makes further refactoring and evolving of the compiler difficult
- Analyzer/Transformers need the source as it is
  - Keep comments/syntactical sugars
- No need for symbols to get involved in meta programming

Scala Meta - Architecture (1)



Scala Meta - Architecture (Scala.meta AST)



Scala Meta - Facilities (1)

- Can parse scala code to scala.meta tree
  At runtime:

```
"object Main extends App { println(1) }".parse[Source].get
// scala.meta.Source$SourceImpl = object Main extends App { println(1) }

Input.File("path/myscalafile.scala").parse[Source]
// scala.meta.parsers.Parsed$Success
```

- And aAt compile time using quasiquotes (macros):

```
q"case class User(name: String, age: Int)"
```

- Textual representation of the scala.meta.Tree

```
“foo(bar)".parse[Stat].get.syntax
// “foo(bar)”

“foo(bar)".parse[Stat].get.structure
// “Term.Apply(Term.Name(\”foo\"), Seq(Term.Name(\"bar\")))"
```

Scala Meta - Facilities (2)

- Scala meta has almost all the facilities of scala reflect
  such as: quasiquote, transformer, traverser, etc.
- Quasiquote syntax is different though
  https://github.com/scalameta/scalameta/blob/master/notes/quasiquotes.md
  Example:

Object	`q"..$mods object $name extends $template"`

Many more quasiquotes has been added to support symbol-less architecture

Scala Meta - Using scala.meta

- Could be used as a standalone source-code parser as well as annotation-macro

- Firstly add scala.meta library to your project

```
libraryDependencies += "org.scalameta" %% "scalameta" % "1.4.0"
```

- In case of scala.meta annotation macros add scala.meta-paradise to your compiler-plugins

```
resolvers += Resolver.bintrayIvyRepo("scalameta", "maven")
scalacOptions += "-Xplugin-require:macroparadise"
addCompilerPlugin(
  "org.scalameta" % "paradise" % "4.0.0.142" cross CrossVersion.full)
```

Scala Meta - Using scala.meta - standalone code transformer (1)

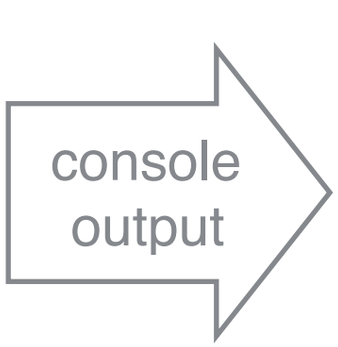
```
import scala.meta._
```

```
def main(args: Array[String]) = {
  val file = Input.File("path/myscalafile.scala")
  val parsed = file.parse[Source].get
  val transformed = MyTransformer(parsed)
  println(transformed.syntax)
}
```

```
object MyTransformer extends Transformer {
  override def apply(tree: Tree): Tree = {
    tree match {
      case q"val $v: $tpe = $value" =>
       val vName = Lit(v.name.toString)
        val replaceWith =
          q"""
              val $v: $tpe = {
                val tmp = $value;
                println("Variable " + $vName + " is initialized with " + tmp);
                tmp
              }
            """
        replaceWith
      case _ =>
        super.apply(tree)
    }
  }
}
```

Scala Meta - Using scala.meta - standalone code transformer (2)

```
object Nima {

  def main(args: Array[String]): Unit = {
    val x: Int = 100
    val y: Int = 200
    val z: Int = 300

    println(s"result: $x")
  }

}
```




```
object Nima {
  def main(args: Array[String]): Unit = {
    val x: Int = {
      val tmp = 100
      println("Variable " + "x" +
        " is initialized with " + tmp)
      tmp
    }
    val y: Int = {
      val tmp = 200
      println("Variable " + "y" +
        " is initialized with " + tmp)
      tmp
    }
    val z: Int = {
      val tmp = 300
      println("Variable " + "z" +
        " is initialized with " + tmp)
      tmp
    }
    println(s"result: $x")
  }
}
```

Scala Meta - Using scala.meta - annotation macro (1)

```
import scala.meta._
```

```
class heavyLog extends scala.annotation.StaticAnnotation {
  inline def apply(tree: Any): Any = meta {
    MyTransformer(tree).asInstanceOf[Stat]
  }
}
```

```
@heavyLog
def main(args: Array[String]): Unit = {
  val x: Int = 100
  val y: Int = 200
  val z: Int = 300
  println(s"result: $x")
}
```

console output

```
Variable x is initialized with 100
Variable y is initialized with 200
Variable z is initialized with 300
result: 100
```

Scala Meta - Using scala.meta - annotation macro (2)

- Amazing tool support by IDEA (2017.1 EAP)

Simple macro I wrote that adds **sayHello** method to an object

```
@addHello
object ClassGeneratorSample extends App {
  this.sayHello
}
```

IDEA can expands your macro

```
object ClassGeneratorSample extends App() {
  def sayHello = println("Hello Scala !")
  this.sayHello
}
```

Deepest expression of appreciation goes to:

Amir Karimi

Sepideh Sabour

Thanks for Listening !

Download sample codes from
https://github.com/nimatrueway/scala-metaprogramming.git

Stay in touch

@nimatrueway