Scala macros

Swiss army knife for building libraries

About me

Bartosz Bąbol

Twitter: @BBartosz91
Blog: www.bbartosz.com

Macros

Scala Documentation API Learn Quickref Contribute SIPs/SLIPs Search in documentation...

MACROS

Def Macros

English 日本語

Eugene Burmako EXPERIMENTAL

Def macros are shipped as an experimental feature of Scala since version 2.10.0. A subset of def macros, pending a thorough specification, is tentatively scheduled to become stable in one of the future versions of Scala.

UPDATE This guide has been written for Scala 2.10.0, and now we're well into the Scala 2.11.x release cycle, so naturally the contents of the document are outdated. Nevertheless, this guide is not obsolete - everything written here will still work in both Scala 2.10.x and Scala 2.11.x, so it will be helpful to read it through. After reading the guide, take a look at the docs on quasiquotes and macro bundles to familiarize yourself with latest developments that dramatically simplify writing macros. Then it might be a good idea to follow our macro workshop for more in-depth examples.

Intuition

Here is a prototypical macro definition:

```
1.  def m(x: T): R = macro implRef
```

At first glance macro definitions are equivalent to normal function definitions, except for their body, which starts with the conditional keyword `macro` and is followed by a possibly qualified identifier that refers to a static macro implementation method.

If, during type-checking, the compiler encounters an application of the macro `m(args)`, it will expand that application by invoking the corresponding macro implementation method, with the abstract-syntax trees of the argument expressions args as arguments. The result of the macro implementation is another abstract syntax tree, which will be inlined at the call site and will be type-checked in turn.

Contents

What are macros good for?

- Code generation
- Static checks
- Domain-specific languages

What are macros?

- allow metaprogramming in Scala
- expanded at compile time
- manipulate AST
- great for building libraries
- written in Scala

What is metaprogramming?

What is metaprogramming?

Metaprogramming in other languages

- C / C++
- Java
- Ruby
- Python
- JavaScript
- Lisp
- Clojure
- Haskell

C macro examples

- Code simplification

```
#define foreach(list, index) \
        for(index = 0; index < list.size(); index++)
```

- Conditional compilation

```
#define _UNIX_
#ifdef _WINDOWS_
#include <windows.h>
#else
#include <pthread.h>
#endif
```

C macro examples

- Happy debugging!

```
#define TRUE FALSE
```

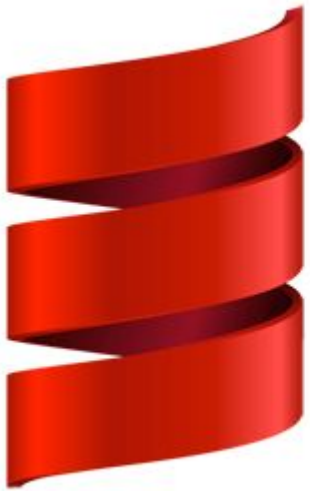
C macro examples

- Happy debugging!

```
#define TRUE FALSE
```

Scala libraries that uses macros

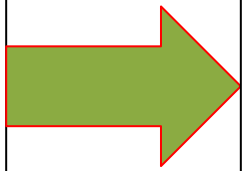
- Play Framework
- Spray
- Shapeless
- Cats
- Slick
- ReactiveMongo
- Rapture



Scala

Motivation

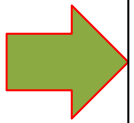
```
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```



```
{
  “year”: 123,
  “model”: “Ford”,
  “weight”: 1000,
  “owner”: “John”
  “color”: “black”
}
```

Motivation

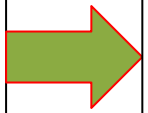
```
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```



```
implicit val carWrites = (
  (__ \ "year").write[Int] and
  (__ \ "model").write[String] and
  (__ \ "weight").write[Int] and
  (__ \ "owner").write[String] and
  (__ \ "color").write[String]
)(unlift(Car.unapply))
```

Motivation

```scala
case class Car(
    year: Int,
    model: String,
    weight: Int,
    owner: String,
    color: String
)
```



```scala
implicit val carWrites = Json.writes[Car]
```

def writes[A]

```
*   implicit val userWrites = Json.writes[User]
*   // macro-compiler replaces Json.writes[User] by injecting into compile
chain
*   // the exact code you would write yourself. This is strictly equivalent
to:
*   implicit val userWrites = (
*      (__ \ 'name).write[String] and
*      (__ \ 'age).write[Int]
*   )(unlift(User.unapply))
* }}}
```

def writes[A]

```
*   implicit val userWrites = Json.writes[User]
*   // macro-compiler replaces Json.writes[User] by injecting into compile
chain
*   // the exact code you would write yourself. This is strictly equivalent
to:
*   implicit val userWrites = (
*      (__ \ 'name).write[String] and
*      (__ \ 'age).write[Int]
*   )(unlift(User.unapply))
* }}}
```

def writes[A]

Implementation in Json.scala

```
def writes[A] = macro JsMacroImpl.writesImpl[A]
```

Simple example of macros

```
object Main extends App{
    Welcome.hello("Lambda Days")
}
```

Simple example of macros

```
import java.time.Year
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context

object Welcome {
    def hello(name: String): Unit = macro helloImplementation
    def helloImplementation(c: Context)(name: c.Tree): c.Tree = {
        import c.universe._
        val currentYear = Year.now.getValue
        q""" println("Hello " + $name + " " + $currentYear) """
    }
}
```

Simple example of macros

```
import java.time.Year
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context

object Welcome {
    def hello(name: String): Unit = macro helloImplementation
    def helloImplementation(c: Context)(name: c.Tree): c.Tree = {
        import c.universe._
        val currentYear = Year.now.getValue
        q""" println("Hello " + $name + " " + $currentYear) """
    }
}
```

Simple example of macros

```
import java.time.Year
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context

object Welcome {
    def hello(name: String): Unit = macro helloImplementation
    def helloImplementation(c: Context)(name: c.Tree): c.Tree = {
        import c.universe._
        val currentYear = Year.now.getValue
        q""" println("Hello " + $name + " " + $currentYear) """
    }
}
```

Simple example of macros

```
import java.time.Year
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context

object Welcome {
    def hello(name: String): Unit = macro helloImplementation
    def helloImplementation(c: Context)(name: c.Tree): c.Tree = {
        import c.universe._
        val currentYear = Year.now.getValue
        q""" println("Hello " + $name + " " + $currentYear) """
    }
}
```

Simple example of macros

```
import java.time.Year
import scala.language.experimental.macros
import scala.reflect.macros.blackbox.Context

object Welcome {
    def hello(name: String): Unit = macro helloImplementation
    def helloImplementation(c: Context)(name: c.Tree): c.Tree = {
        import c.universe._
        val currentYear = Year.now.getValue
        q""" println("Hello " + $name + " " + $currentYear) """
    }
}
```

Quasiquotes

```
q""" println("Hello " + $name + " " + $currentYear)"""
```

What are AST

JAVEO

1 + (2 + 3)

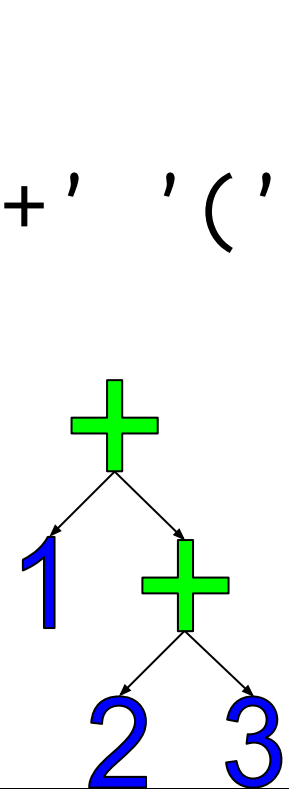
What is Abstract Syntax Tree (AST)

```
1 + (2 + 3)
```

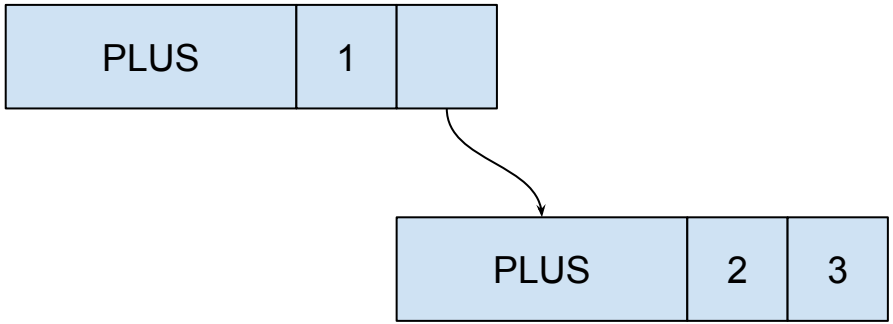
```
int(1) '+' '(' int(2) '+' int(3) ')'
```

What is Abstract Syntax Tree (AST)

```
1 + (2 + 3)
```

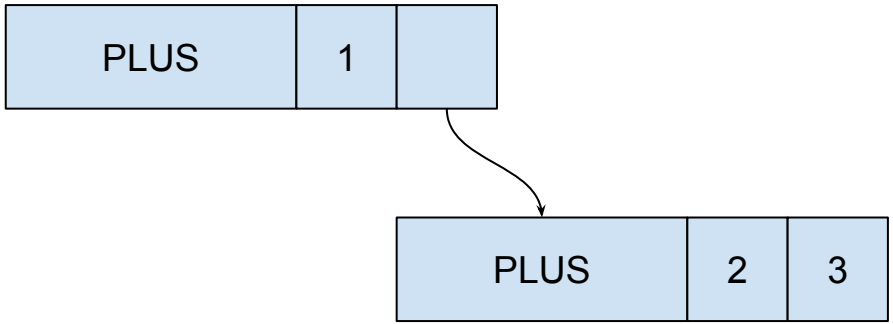


```
int(1) '+' '(' int(2) '+' int(3) ')'
```

```
1 + (2 + 3)
```

```
Plus(1, (Plus(2, 3))
```



Quasiquotes

```
q""" println("Hello " + $name + " " + $currentYear)"""
```

```
Apply(
     Ident(TermName("println")),
     List(
         Apply(
               Select(
               Apply(
                    Select(
                         Apply(
                              Select(
                                   Literal(Constant("Hello ")),
                                   TermName("$plus")),
                              List(
                                   Literal(Constant("Lambda days")))),
                              TermName("$plus")),
                         List(
                              Literal(Constant(" "))))),
                         TermName("$plus")),
                         List(
                              Literal(Constant(2016))))))
```

Before/after quasiquotes

Quasiquotes	Tree
q“”” “lambda days” “””	Literal(Constant("lambda days"))
q”x+y”	Apply(Select(Ident(TermName("x")), TermName("$plus")), List(Ident(TermName("y"))))

Other quasiquotes interpolators

```
val expressionsDefinitionsAndImports = q"List(1,2)"

val types = tq"List[Int]"

val patterns = pq"a: List[Int]"

val caseClause = cq"a: Int => a + 2"

val forLoopEnumerator = fq"a <- 1 to 5"
```

Macro annotations

```
def testMethod[String]: Double = {
  val x = 2.0 + 2.0
  Math.pow(x, x)
}
```

Macro annotations

```
def testMethod[String]: Double = {
  val start = System.nanoTime()
  val x = 2.0 + 2.0
  val result = Math.pow(x, x)
  val end = System.nanoTime()
  println("testMethod elapsed time: " + (end - start) + "ns")
  result
}
```

Macro annotations

```
def testMethod[String]: Double = {
      val start = System.nanoTime()
      val x = 2.0 + 2.0
      val result = Math.pow(x, x)
      val end = System.nanoTime()
      println("testMethod elapsed time: " + (end - start) + "ns")
      result
}
```

Macro annotations

```
@Benchmark
def testMethod[String]: Double = {
      val x = 2.0 + 2.0
      Math.pow(x, x)
}
```

Macro annotations

```
@Benchmark
def testMethod[String]: Double = {
      val x = 2.0 + 2.0
      Math.pow(x, x)
}
```

```
def testMethod[String]: Double = {
      val t0 = System.nanoTime()
      val x = 2.0 + 2.0
      val result = Math.pow(x, x)
      val t1 = System.nanoTime()
      println("testMethod elapsed time:
    " + (t1 - t0) + "ns")
      result
}
```

Macro annotations

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

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

Macro annotations

```
object Benchmark {
    def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
        import c.universe._
        val result = {
            annottees.map(_.tree).toList match {
                case q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
                    :: Nil => {
                    // Creating new AST tree here
                }
                case _ => c.abort(c.enclosingPosition, "Annotation @Benchmark can be used
                                                        only with methods")
            }
    }
    c.Expr[Any](result) }}
```

Macro annotations

```
object Benchmark {
    def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
        import c.universe._
        val result = {
            annottees.map(_.tree).toList match {
                case q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
:: Nil => {
                    // Creating new AST tree here
                }
                case _ => c.abort(
                    c.enclosingPosition,
                    "Annotation @Benchmark can be used only with methods")
            }
    }
    c.Expr[Any](result) }}
```

Macro annotations

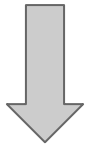
```
object Benchmark {
    def impl(c: Context)(annottees: c.Expr[Any]*): c.Expr[Any] = {
        import c.universe._
        val result = {
            annottees.map(_.tree).toList match {
                case q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
                    :: Nil => {
                    // Creating new AST tree here
                }
                case _ => c.abort(c.enclosingPosition, "Annotation @Benchmark can be used
                                                        only with methods")
            }
    }
    c.Expr[Any](result)
    }}
```

Macro annotations

```
q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
```

Macro annotations

```
q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
```

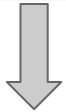


```
..$tpes   => List[universe.Tree]
...$args => List[List[universe.Tree]]
```

Macro annotations

```
q"$mods def $methodName[..$tpes](...$args): $returnType = { ..$body }"
```

```
..$tpes   => List[universe.Tree]
...$args => List[List[universe.Tree]]
```



```
private def foo[A, B, C](a:A ,b:B)(c: C): A = {
  //body
}
```

Macro annotations

```
q"""$mods def $methodName[..$tpes](...$args): $returnType =  {
    val start = System.nanoTime()
    val result = {..$body}
    val end = System.nanoTime()
    println(${methodName.toString} + " elapsed time: " + (end - start) +
    "ns")
    result
}"""
```

Gridzzly

GRIDZZLY

Our case

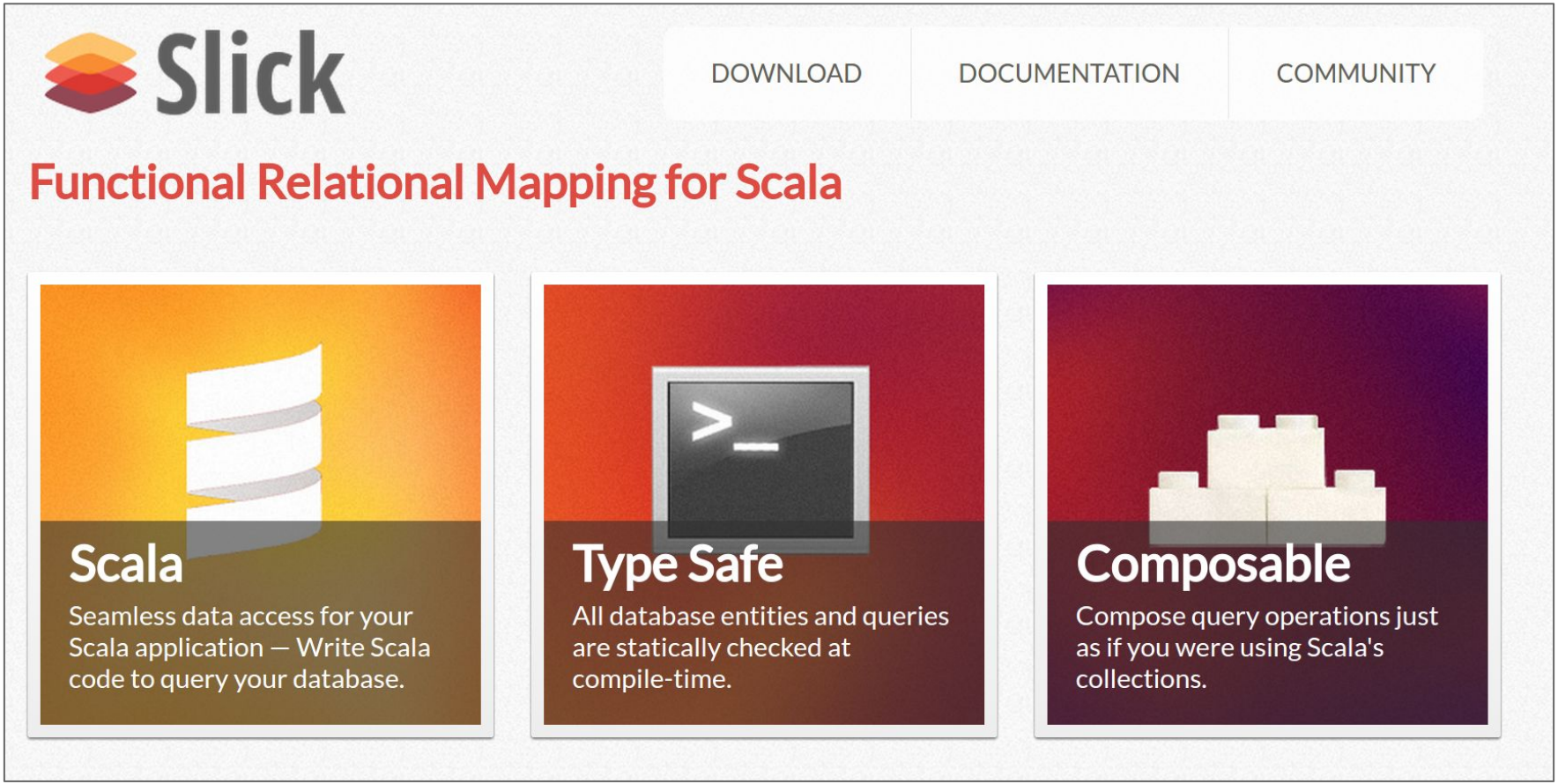
Users list

Generate 120 users

#	Name	Last name	Status	Gender	Email	Notes	Car Brand
	Search	Search	All	All	Search	Search	Search
1	Braulio	Lind	Blocked	female	edwina.reichel@cummings.biz	Iure enim dolores dolores quisquam et aut ea laboriosam.	
2	Nichole	Yost	Premium	male	mitchel_harber@little.net	Tenetur enim voluptatem illum voluptatem rerum ex pariatur.	
3	Ericka	Paucek	Premium	male	robbie_mcglynn@bernier.name	Qui dolores numquam rerum blanditiis labore vel quos qui inventore.	
4	Albertha	Funk	Active	female	alexandria@faheylarson.org	Ad iste aliquid mollitia impedit culpa fugiat inventore et.	
5	Dusty	Schuppe	Premium	male	pete_reinger@greenokeefe.biz	Quia provident ut fuga saepe repellat.	
6	Jeremy	Schuster	Blocked	female	kelvin@kovacek.info	Qui molestiae aut quibusdam repudiandae vero.	
7	Arlo	Rath	Blocked	male	verona@schulist.org	Facere excepturi et vel dolor iure quasi vitae facere dolor expedita.	
8	Natalie	Baumbach	Active	female	delbert_kiehn@stiedemann.us	Nemo culpa maiores hic omnis fuga autem omnis.	
9	Ellis	Ryan	Active	male	edmond@prohaskalubowitz.com	Dolores nisi praesentium omnis ut vitae libero.	
10	Kyle	Rohan	Active	male	delia_kertzmann@beahan.info	Libero quis earum est quia et facilis voluptas.	

1 2 3 4 5

Slick


Slick
DOWNLOAD
DOCUMENTATION
COMMUNITY
Functional Relational Mapping for Scala
Scala
Seamless data access for your Scala application — Write Scala code to query your database.
Type Safe
All database entities and queries are statically checked at compile-time.
Composable
Compose query operations just as if you were using Scala's collections.

Slick

```
users.map(_.name)    =>  select NAME from USERS
```

Slick

```
class UsersTable(tag: Tag) extends Table[User](tag, "users"){
    def id = column[Long]("id", O.PrimaryKey, O.AutoInc)
    def email = column[String]("email")
    def firstName = column[String]("first_name")
    def lastName = column[String]("last_name")
    def status = column[Int]("status")
    def gender = column[Int]("gender")
    def salary = column[Long]("salary")
    def notes = column[String]("notes")

    def * = (id.?, email, firstName, lastName, status, gender,salary, notes)
            <> ((User.apply _).tupled, User.unapply)
}
```

Slick

```
val filtered = users.filter(_.email === “a@a.pl”) =>
select *
from "users"
where "email" = "a@a.pl"
```

Filtering

```
val paginated = users.drop(10).take(5)
select *
from "users"
limit 5 offset 10
```

Paginating

```
val sorted = users.sortBy(_.firstName.desc)
select *
from "users"
order by "first_name" desc
```

Sorting

Sorting

```
def sortBy(conditions: (String, String)) = conditions match {
 case ("firstName", "asc") => users.sortBy{case user => user.firstName.asc}
 case ("firstName", "desc") => users.sortBy{ case user => user.firstName.desc}
 case ("lastName", "asc") => users.sortBy{ case user => user.lastName.asc}
 case ("lastName", "desc") => users.sortBy{ case user => user.lastName.desc}
 case ("status", "asc") => users.sortBy{ case user => user.status.asc}
 case ("status", "desc") => users.sortBy{ case user => user.status.desc}
 case ("gender", "asc") => users.sortBy{ case user => user.gender.asc}
 case ("gender", "desc") => users.sortBy{ case user => user.gender.desc}
 case ("notes", "asc") => users.sortBy{ case user => user.notes.asc}
 case ("notes", "desc") => users.sortBy{ case user => user.notes.desc}
 case ("salary", "asc") => users.sortBy{ case user => user.salary.asc}
 case ("salary", "desc") => users.sortBy{ case user => user.salary.desc}
 case ("email", "asc") => users.sortBy{ case user => user.email.asc}
 case ("email", "desc") => users.sortBy{ case user => user.email.desc}
 case _ => users.sortBy{ case user => user.id.desc} }
```

Slick

```
def sortBy(conditions: (String, String)) = conditions match {
 case ("firstName", "asc") => users.sortBy{case user => user.firstName.asc}
 case ("firstName", "desc") => users.sortBy{ case user => user.firstName.desc}
 case ("lastName", "asc") => users.sortBy{ case user => user.lastName.asc}
 case ("lastName", "desc") => users.sortBy{ case user => user.lastName.desc}
 case ("status", "asc") => users.sortBy{ case user => user.status.asc}
 case ("status", "desc") => users.sortBy{ case user => user.status.desc}
 case ("gender", "asc") => users.sortBy{ case user => user.gender.asc}
 case ("gender", "desc") => users.sortBy{ case user => user.gender.desc}
 case ("notes", "asc") => users.sortBy{ case user => user.notes.asc}
 case ("notes", "desc") => users.sortBy{ case user => user.notes.desc}
 case ("salary", "asc") => users.sortBy{ case user => user.salary.asc}
 case ("salary", "desc") => users.sortBy{ case user => user.salary.desc}
 case ("email", "asc") => users.sortBy{ case user => user.email.asc}
 case ("email", "desc") => users.sortBy{ case user => user.email.desc}
 case _ => users.sortBy{ case user => user.id.desc} }
```

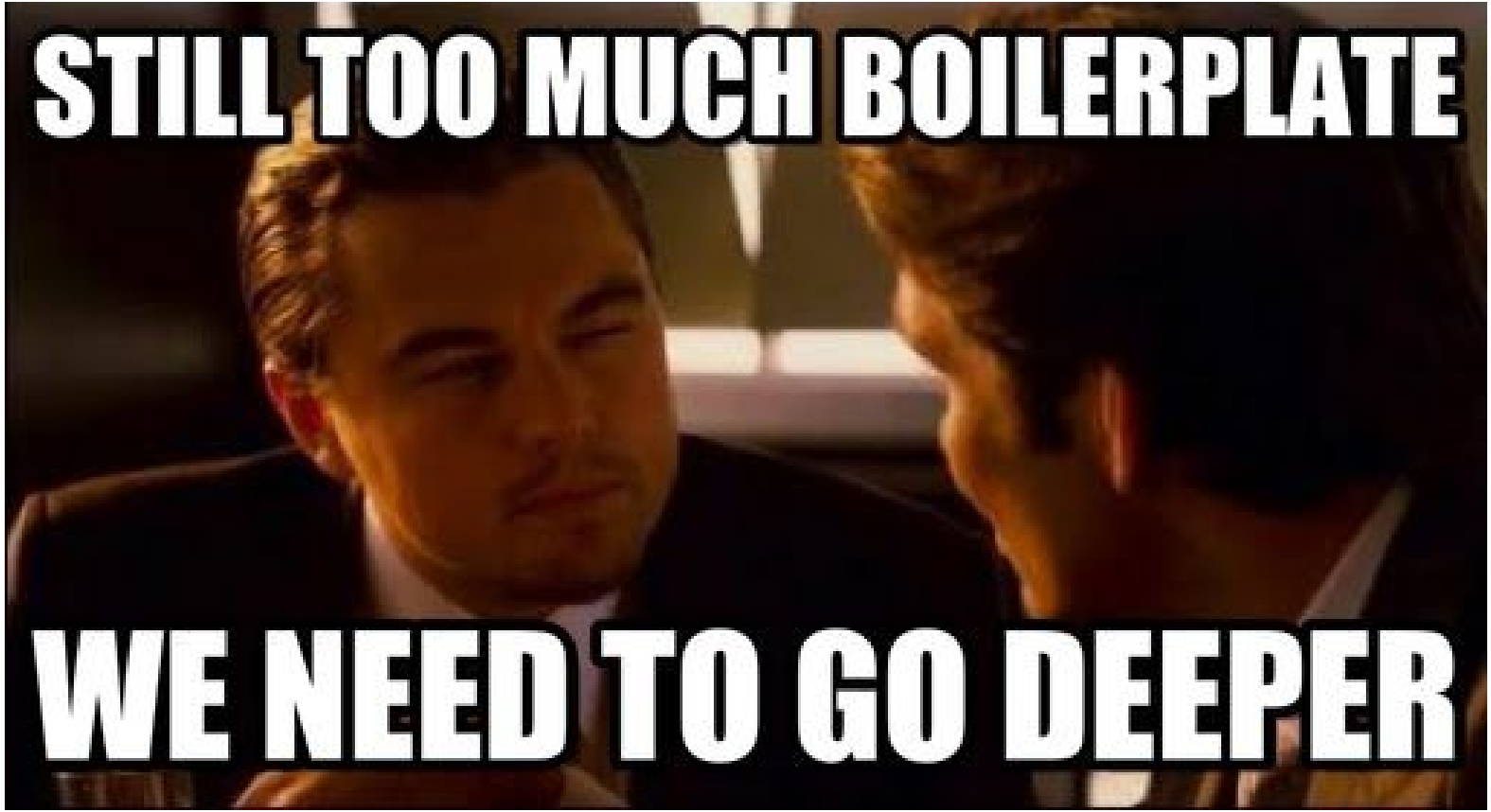
SortBy

```
def sortBy(conditions: (String, String)) = conditions match {
 case ("firstName", "asc") => users.sortBy{case user => user.firstName.asc}
 case ("firstName", "desc") => users.sortBy{ case user => user.firstName.desc}
 case ("lastName", "asc") => users.sortBy{ case user => user.lastName.asc}
 case ("lastName", "desc") => users.sortBy{ case user => user.lastName.desc}
 case ("status", "asc") => users.sortBy{ case user => user.status.asc}
 case ("status", "desc") => users.sortBy{ case user => user.status.desc}
 case ("gender", "asc") => users.sortBy{ case user => user.gender.asc}
 case ("gender", "desc") => users.sortBy{ case user => user.gender.desc}
 case ("notes", "asc") => users.sortBy{ case user => user.notes.asc}
 case ("notes", "desc") => users.sortBy{ case user => user.notes.desc}
 case ("salary", "asc") => users.sortBy{ case user => user.salary.asc}
 case ("salary", "desc") => users.sortBy{ case user => user.salary.desc}
 case ("email", "asc") => users.sortBy{ case user => user.email.asc}
 case ("email", "desc") => users.sortBy{ case user => user.email.desc}
 case _ => users.sortBy{ case user => user.id.desc} }
```

SortBy

```
def sortBy(conditions: (String, String)) = conditions match {
 case ("firstName", "asc") => users.sortBy{case user => user.firstName.asc}
 case ("lastName", "asc") => users.sortBy{ case user => user.lastName.asc}
 case ("status", "asc") => users.sortBy{ case user => user.status.asc}
 case ("gender", "asc") => users.sortBy{ case user => user.gender.asc}
 case ("notes", "asc") => users.sortBy{ case user => user.notes.asc}
 case ("salary", "asc") => users.sortBy{ case user => user.salary.asc}
 case ("email", "asc") => users.sortBy{ case user => user.email.asc}
 case _ => users.sortBy{ case user => user.id.desc} }
```

SortBy



SortBy

```
def sortBy(conditions: (String, String)) = conditions match {
 case ("firstName", "asc") => users.sortBy{case user => user.firstName.asc}
 case ("lastName", "asc") => users.sortBy{ case user => user.lastName.asc}
 case ("status", "asc") => users.sortBy{ case user => user.status.asc}
 case ("gender", "asc") => users.sortBy{ case user => user.gender.asc}
 case ("notes", "asc") => users.sortBy{ case user => user.notes.asc}
 case ("salary", "asc") => users.sortBy{ case user => user.salary.asc}
 case ("email", "asc") => users.sortBy{ case user => user.email.asc}
 case _ => users.sortBy{ case user => user.id.desc} }
```

FilterBy

```
private def filter(query: MyQuery, filterBy: Map[String, String]): MyQuery =   query.filter({
 case (user, car) => List(
   Option(filterBy.getOrElse("firstName")).map(col => user.firstName.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("lastName")).map(col => user.lastName.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("status")).map(col => user.status.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("gender")).map(col => user.gender.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("notes")).map(col => user.notes.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("salary")).map(col => user.salary.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("email")).map(col => user.email.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%"))).collect({
   case Some((criteria)) => criteria
 }).reduceLeftOption(_ || _).getOrElse((true: Rep[Boolean]))
})
```

FilterBy

```
private def filter(query: MyQuery, filterBy: Map[String, String]): MyQuery =   query.filter({
 case (user, car) => List(
   Option(filterBy.getOrElse("firstName")).map(col => user.firstName.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("lastName")).map(col => user.lastName.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("status")).map(col => user.status.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("gender")).map(col => user.gender.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("notes")).map(col => user.notes.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("salary")).map(col => user.salary.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%")),
   Option(filterBy.getOrElse("email")).map(col => user.email.asColumnOf[String].toLowerCase.like("%"+col.toLowerCase+"%"))).collect({
   case Some((criteria)) => criteria
 }).reduceLeftOption(_ || _).getOrElse((true: Rep[Boolean]))
})
```

Gridzzly syntax

```
@Gridzzly
case class UsersGrid() extends Grid[UsersTable, User, Seq]{
 val query = for {
   user <- TableQuery[UsersTable]
 } yield user

 val columns = Seq(
   GridColumn[UsersTable, String]("First name", user => user.firstName),
   GridColumn[UsersTable, String]("Last name", user => user.lastName),
   GridColumn[UsersTable, Int]("Status", user => user.status, Equals()),
   GridColumn[UsersTable, Int]("Gender", user => user.gender, Equals()),
   GridColumn[UsersTable, String]("Notes", user => user.notes),
   GridColumn[UsersTable, Long]("Salary", user => user.salary),
   GridColumn[UsersTable, String]("Email", user => user.email))

 val defaultSortBy = DefaultGridColumn[UsersTable, Long](user => user.id.desc)
}
```

```
@Gridzzly
case class UsersGrid() extends Grid[UsersTable, User, Seq]{
 val query = for {
   user <- TableQuery[UsersTable]
 } yield user

 val columns = Seq(
   GridColumn[UsersTable, String]("First name", user => user.firstName),
   GridColumn[UsersTable, String]("Last name", user => user.lastName),
   GridColumn[UsersTable, Int]("Status", user => user.status, Equals()),
   GridColumn[UsersTable, Int]("Gender", user => user.gender, Equals()),
   GridColumn[UsersTable, String]("Notes", user => user.notes),
   GridColumn[UsersTable, Long]("Salary", user => user.salary),
   GridColumn[UsersTable, String]("Email", user => user.email))

 val defaultSortBy = DefaultGridColumn[UsersTable, Long](user => user.id.desc)
}
```

```
case class UsersGrid() extends Grid[UsersTable, User, Seq] {
 ...
 type MyQuery = slick.lifted.Query[scala.Tuple2[UsersTable, Rep[Option[CarsTable]]], scala.Tuple2[User, Option[Car]], Seq];
 def run(conditions: GridConditions)(implicit dbConnection: DBConnection) = {
  val filteredQuery = filter(query, conditions.filterBy.columns)
  val sorted = sortBy(conditions, filteredQuery).drop((conditions.page - 1) * conditions.perPage).take(conditions.perPage).result
  val count = countFiltered(conditions).result
  (dbConnection.db.run(sorted), dbConnection.db.run(count))
 }
 def run(conditions: GridConditions, initialFilter: ((UsersTable, Rep[Option[CarsTable]])) => Rep[Boolean] )(implicit dbConnection:
DBConnection) = {
  val initialFiltered = query.filter(initialFilter)
  val filteredQuery = filter(initialFiltered, conditions.filterBy.columns)
  val sorted = sortBy(conditions, filteredQuery).drop((conditions.page - 1) * conditions.perPage).take(conditions.perPage).result
  val count = countInitiallyFiltered(conditions, initialFiltered).result
  (dbConnection.db.run(sorted), dbConnection.db.run(count))
 }

 private def countInitiallyFiltered(conditions: GridConditions, filteredQuery: MyQuery) = filter(filteredQuery, conditions.filterBy.
columns).length;

 val colsForFrontend = scala.collection.immutable.List(
  scala.Tuple2("First name", "user.firstName"), scala.Tuple2("Last name", "user.lastName"),
  scala.Tuple2("Status", "user.status"), scala.Tuple2("Gender", "user.gender"), scala.Tuple2("Notes", "user.notes"),
  scala.Tuple2("Salary", "user.salary"), scala.Tuple2("Car brand", "carBrand"), scala.Tuple2("Email", "user.email")).map({
  case scala.Tuple2((label), (name)) => ColForFrontend(label, name)
 })
 private def countFiltered(conditions: GridConditions) = filter(query, conditions.filterBy.columns).length;

 private def sortBy(conditions: GridConditions, query: MyQuery) = conditions match {
  case GridConditions(_, _, "user.firstName", "asc", _) => query.sortBy({ case scala.Tuple2((user), (car)) => user.firstName.asc
  })
  case GridConditions(_, _, "user.firstName", "desc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.firstName.desc
  })
  case GridConditions(_, _, "user.lastName", "asc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.lastName.asc
  })
  case GridConditions(_, _, "user.lastName", "desc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.lastName.desc
  })
  case GridConditions(_, _, "user.status", "asc", _) => query.sortBy(case scala.Tuple2((user), (car)) => user.status.asc
  })
  case GridConditions(_, _, "user.status", "desc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.status.desc
  })
  case GridConditions(_, _, "user.gender", "asc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.gender.asc
  })
  case GridConditions(_, _, "user.gender", "desc", _) => query.sortBy({case scala.Tuple2((user), (car)) => user.gender.desc
  })
```

Gridzzly syntax

```
@Gridzzly
case class UsersGrid() extends Grid[UsersTable, User, Seq]{
 val query = for {
   user <- TableQuery[UsersTable]
 } yield user

 val columns = Seq(
   GridColumn[UsersTable, String]("First name", user => user.firstName),
   GridColumn[UsersTable, String]("Last name", user => user.lastName),
   GridColumn[UsersTable, Int]("Status", user => user.status, Equals()),
   GridColumn[UsersTable, Int]("Gender", user => user.gender, Equals()),
   GridColumn[UsersTable, String]("Notes", user => user.notes),
   GridColumn[UsersTable, Long]("Salary", user => user.salary),
   GridColumn[UsersTable, String]("Email", user => user.email))

 val defaultSortBy = DefaultGridColumn[UsersTable, Long](user => user.id.desc)
}
```

Gridzzly syntax

```
@Gridzzly
case class UsersGrid() extends Grid[UsersTable, User, Seq]{
 val query = for {
   user <- TableQuery[UsersTable]
 } yield user

 val columns = Seq(
   GridColumn[UsersTable, String]("First name", user => user.firstName),
   GridColumn[UsersTable, String]("Last name", user => user.lastName),
   GridColumn[UsersTable, Int]("Status", user => user.status, Equals()),
   GridColumn[UsersTable, Int]("Gender", user => user.gender, Equals()),
   GridColumn[UsersTable, String]("Notes", user => user.notes),
   GridColumn[UsersTable, Long]("Salary", user => user.salary),
   GridColumn[UsersTable, String]("Email", user => user.email))

 val defaultSortBy = DefaultGridColumn[UsersTable, Long](user => user.id.desc)
}
```

Gridzzly syntax

```
@Gridzzly
case class UsersGrid() extends Grid[UsersTable, User, Seq]{
 val query = for {
   user <- TableQuery[UsersTable]
 } yield user

 val columns = Seq(
   GridColumn[UsersTable, String]("First name", user => user.firstName),
   GridColumn[UsersTable, String]("Last name", user => user.lastName),
   GridColumn[UsersTable, Int]("Status", user => user.status, Equals()),
   GridColumn[UsersTable, Int]("Gender", user => user.gender, Equals()),
   GridColumn[UsersTable, String]("Notes", user => user.notes),
   GridColumn[UsersTable, Long]("Salary", user => user.salary),
   GridColumn[UsersTable, String]("Email", user => user.email))

 val defaultSortBy = DefaultGridColumn[UsersTable, Long](user => user.id.desc)
}
```

IDE support

```
def list(page: Int, perPage: Int, sortBy: String, sortDir: String, filterBy: FilterColumns) = Action.async { implicit request =>
  val gridConditions = GridConditions(page, perPage, sortBy, sortDir, filterBy)
  val searchResults = UsersGrid().run(gridConditions)

  val result = for {
    users <- searchResults._1
    count <- searchResults._2
  } yield UsersDto(users.map{case (user, car) => UserWithCarDto(user.toUserDto, car)}, count)

  result.map(response => Ok(Json.toJson(response)))
}
```

IDE Support

IntellijIDEA

Intellij Idea support for macros

← IntelliJ IDEA 15 EAP Adds Postfix Code Completion for Scala

Scala Plugin EAP Speeds Up Coding Assistance →

IntelliJ API to Build Scala Macros Support

Posted on October 14, 2015 by Andrey Cheptsov

Today we've released a new Scala plugin EAP build. With this build, we're happy to introduce an API that extends the IDE coding assistance to custom Scala macros.

A macro is executed at compile-time and modifies the AST of your code: e.g. it can extend the class or its companion object with new methods or other code represented by the AST returned by the macro. Since IntelliJ IDEA's coding assistance is based on static code

Debugging

Thank you for listening!

Where to find more?

http://scalamacros.org/talks.html

http://underscore.io/training/courses/essential-macros/

Gridzzly:

https://github.com/JAVEO/gridzzly

https://github.com/JAVEO/gridzzly-example

https://github.com/JAVEO/gridzzly-idea-plugin

And last but not least:

bbartosz.com