Рефлексия и макросы

Рефлексия и макросы

В этой серии лекций мы познакомимся с рефлексией и макросами.
Рефлексия - это набор средств, с помощью которых приложение способно исследовать само себя, как на этапе компиляции так и в процессе выполнения.
Макросы - это подпрограммы, преобразующие код основного приложения.
Рефлексия и макросы широко используются для тестирования, логирования, профилирования приложений и т.д. Многие популярные скала библиотеки (например **shapless**, **cats**, **mockito**) были бы невозможны без этих механизмов
Краткое содержание раздела

- Требования, причины возникновения и подготовка к использованию
- Термины и понятия
- Рефлексия времени выполнения
- Рефлексия времени компиляции
- Макросы
  - blackbox
  - whitebox
  - annotations

Рефлексия и макросы

Рефлексия
С помощью рефлексии можно можно сделать очень многое.

- получить информацию о любых, даже приватных, членах любого класса, трейта, объекта и т. д
- узнать иерархию типов в рантайме
- сохранить информацию о типах, которая теряется при запуске приложения из-за **type erasure**
- динамически загружать новые классы
- создавать инстансы и вызывать методы произвольных классов на лету
- и даже компилировать и запускать код из обычного текста

Рефлексия в scala представлена как собственными механизмами, так и механизмами, пришедшими из java. Собственный reflection API для скала потребовался в первую очередь, для того, чтобы предоставить удобный механизм работы с теми аспектами языка которых нет в java. Например, с трейтами, кейс классами ,специфическим описанием констант и переменных, более сложным параметрическим полиморфизмом и т.д.

Изучение рефлексии начнем с “классических” механизмов, доступных, как в java так и в scala.

Рефлексия и макросы

Java рефлексия
При старте во время выполнения java (и scala) приложения информация о класса загружается в память виртуальной машины. За загрузку классов отвечает один или несколько наследников абстрактного класса **java.lang.ClassLoader.** По мере загрузки информация о классах сохранятся в инстансах класса **java.lang.Class**. Практически любая работа с рефлексией начинается с получения инстанса **Class** и с получения доступа к нужному классслоадеру. Кроме перечисленных классов, java рефлекшен включает в себя еще несколько полезных вспомогательных типов. Вот некоторые из них:

- **java.lang.reflect.Member -** родительский класс для всех классов, описывающих члены классов
- **java.lang.reflect.Field -** тип позволяющий получить информацию о типе члена класса, а также получять и устанавливать значение поля на данном инстансе типа.
- **java.lang.reflect.Method -** описывает методы класса и позволяет их вызывать
- **java.lang.reflect.Constructor -** содержит описание конструктора типа. Вызов инстанса **Constructor,** создает новый инстанс описываемого типа
- **java.lang.reflect.Modifier -** это информация о модификаторах доступа к членам класса и к самим классам.

Рефлексия и макросы

Java рефлексия.
Получить инстанс Class можно несколькими способами. Предположим, что у нас есть следующая иерархия классов

```
trait JavaReflectExampleTrait {
 protected val field: Int
 private var privateField: String = ""
 val publicField: Long = 0
 def identity(): Unit = ()
}

object JavaReflectExampleObject extends JavaReflectExampleTrait {
 override protected val field: Int = 30
}

class JavaReflectExample extends JavaReflectExampleTrait {
 override val field: Int = 20
}
```

Рефлексия и макросы

Java рефлексия.
Нам доступны следующие способы:

```
// анализ полей и членов классов
val jert1 = new JavaReflectExampleTrait(){
 override protected val field: Int = 100
}
// получение класса по инстансу
val jert1Cl = jert1.getClass
// по типу объекта
val jreTCl = BigDecimal.getClass // classOf[BigDecimal]
// по строковому имени тип
val jreCl1 = Class.forName("lectures.reflection.JavaReflectExample")
```

Стоит обратить внимание на то, что **jert1Cl** будет иметь имя **lectures.reflection.PlainJavaReflection$$anon$1.** Это связано с тем, что Java рефлекшен не умеет отображать трейты скалы. Он способен показать лишь их представление в виде джава классов.

Рефлексия и макросы

Java рефлексия.
Синтаксис **getClass** имеет ограничение. Оно связано с тем, что массивы не загружаются с помощью классloадера и соответственно не имеют инстанса типа **Class**. **gc** в примере ниже будет иметь значение **Null**

```
val array: Array[Int] = Array(1,2,3,4)
// gc здесь буде равен Null
val gc = array.getClass
```

ClassLoader
Любая работа в рантайме с типами в Java(и в scala соответственно) начинается с загрузки описания класса. Для этого используются классloадеры. Они образуют древовидную структуру. Перед тем, как попробовать загрузить класс, классloадер проверяет, загружен ли уже этот класс родительским классloадером. Классloадер может быть один, как в примере ниже, так и много. Например, приложения, загружаемые в контейнеры сервлетов, часто имеют несколько классloадеров. Классloадеры могут загружать class файлы, которые находятся в локальной файловой системе или загружать описания по сети, например с помощью **URLClassLoader**

Рефлексия и макросы

Java рефлексия. ClassLoader

```
//получение класслоадера
val loaderFroBigDecimal = jreTCl.getClassLoader
val loaderForJavaReflectionExample = jreCl1.getClassLoader
val systemClassLoader = ClassLoader.getSystemClassLoader

val thread = new Thread{
  override def start(): Unit = {
    super.start()
    val threadContextClassLoader = Thread.currentThread().getContextClassLoader() //is the key !
    print(threadContextClassLoader)
  }
}
thread.start()
```

- **ClassLoader.getSystemClassLoader** вернет корневой класслоадер
- **jreCl1.getClassLoader -** это класслоадер, которым был загружен конкретный класс
- **Thread.currentThread().getContextClassLoader() -** этот класслоадер был передан из ThreadFactory, которой был создан поток

Рефлексия и макросы

Java рефлексия. Resources
С спомощью **Class** и **ClassLoader** мы можем получить доступ к различным ресурсам. Вызов метода **getResource** или **getResourceAsStream** вернет ресурсы относительно пакета класс. Вызов этих же методов у классслоадера, вернут ресурсы относительно корневой директории классслоадера

```
// ресурсы относительно классслоадера и класса
val forNameResource: URL = classForName.getResource("forName.txt")
val getClassResource: URL  = this.getClass().getResource("forName.txt")
val relativeToClassInstance: URL  = this.getClass().getResource("./forName.txt")
val relativeToClassLoader: URL = Thread.currentThread().getContextClassLoader().getResource("forContext.txt")
val relativeToClassLoader1: URL = classForName.getClassLoader.getResource("forContext.txt")
```

Рефлексия и макросы

Java рефлексия. Доступ к членам класса

Методы Class для доступа у информации о полях

Class API	Список?	Наследованные поля?	Приватные члены?
getDeclaredField()	no	no	yes
getField()	no	yes	no
getDeclaredFields()	yes	no	yes
getFields()	yes	yes	no

Рефлексия и макросы

Java рефлексия. Доступ к членам класса

Методы Class для доступа к информации о методах

Class API	Список членов?	Наследованые члены?	Приватные члены?
getDeclaredMethod()	no	no	yes
getMethod()	no	yes	no
getDeclaredMethods()	yes	no	yes
getMethods()	yes	yes	no

Рефлексия и макросы

Java рефлексия. Доступ к членам класса

Методы Class для доступа к информации о конструкторах

Class API	Список?	Наследованные члены?	Приватные методы?
getDeclaredConstructor()	no	N/A[1]	yes
getConstructor()	no	N/A[1]	no
getDeclaredConstructors()	yes	N/A[1]	yes
getConstructors()	yes	N/A[1]	no

1- конструкторы не наследуются

Рефлексия и макросы

Java рефлексия. Доступ к членам класса

Методы Class для доступа к информации о конструкторах

Class API	Список?	Наследованные члены?	Приватные методы?
getDeclaredConstructor()	no	N/A[1]	yes
getConstructor()	no	N/A[1]	no
getDeclaredConstructors()	yes	N/A[1]	yes
getConstructors()	yes	N/A[1]	no

1- конструкторы не наследуются

Рефлексия и макросы

Java рефлексия. Доступ к членам класса
Пример применения методов находится в **RetrievingClassInfoWithJava.scala**

Вызов методов
java.lang.reflect.Method обладает методом **invoke**. Первым параметром этого метода является инстанс на котором нужно вызвать метод или null, если метод вызывается у объекта. Остальный параметры - это сисок переменной длинны, содержаший аргумены с которыми надо вызвать метод. Если необходимо вызвать приватный метод, перед вызовом необходимо установить флаг **accessible** в **true** с помощью метода **setAccessible**

```
val inst =  new JavaReflectExample()
val m = classOf[JavaReflectExample].getDeclaredMethod("identity", classOf[Any])
m.setAccessible(true)
val o = m.invoke(inst, Seq(10))
```

Рефлексия и макросы

Java рефлексия, установка значений полей

Установить значение поля можно использовав метод **set**, класса **java.lang.reflect.Field**

```
val field = clazz.getDeclaredField("field")
field.setAccessible(true)
field.set(inst, 40)
val afterSet = inst.field
```

Рефлексия и макросы

Java рефлексия, создание новых инстансов.

Существует 2 рефлективных метода создания инстансов классов: **java.lang.reflect.Constructor.newInstance()** and **Class.newInstance()**. Первый из них предпочтительнее потому что:

- **Class.newInstance()** может вызывать только конструктор без параметров в отличии от **Constructor.newInstance()**.
- **Class.newInstance()** выбрасывает наружу любые исключения случившиеся в процессе работы конструктора. **Constructor.newInstance()** всегда оборачивает исключения в **InvocationTargetException**.
- **Class.newInstance()** не может вызывать недоступные конструкторы; **Constructor.newInstance()**может вызывать недоступные конструкторы в некоторых случаях.

```
val const = clazz.getConstructor(Seq[Class[_]](): _*)
val reflectiveInst = const.newInstance()
```

Все примеры этого раздела можно найти в **lectures.reflection.SettingInvokingAndCreatingWithJava.scala**

Рефлексия и макросы

Scala reflection
С развитием scala стало очевидно, что функций java рефлексии недостаточно, а теми, что есть не всегда удобно пользоваться. Поэтому начиная с версии 2.10 scala свою собственную библиотеку. Чтобы ею воспользоваться нужно добавить ее в зависимости проекта. Например для sbt это можно сделать вот так **libraryDependencies += "org.scala-lang" % "scala-reflect" % "yourVersion"**
Первое важное отличие scala reflection от java - наличие **runtime** и **compiletime** рефлексии. Runtime рефлексия по-сути похожа на рефлексию в java. Compiletime рефлексия - это набор библиотек для генерации кода на этапе компиляции.
Из-за наличия 2-х принципиально отличающихся рефлексий, были введены, так называемые, вселенные, наследницы **scala.reflect.api.Universe**

- **scala.reflect.api.JavaUniverse** отвечает за runtime рефлексию
- **scala.reflect.macros.Universe** отвечает за compiletime рефлексию

Еще одним нововведением является концепция зеркал (Mirrors). Зеркала являются ключевой частью рефлексии. Вся информация о программе, так или иначе доступна через зеркала. Зеркал бывает несколько

- Зеркала, работающие с классами и классслоадерами
- Зеркала, предназначенные для динамической работы с классами. Т.е для вызова методов, создания новых инстансов и т.д. Они доступн только в runtime рефлексии
- Заркала объединяющие 2 предыдущих типа.

Рефлексия и макросы

Scala reflection
Помимо вселенных и зеркал, было введено большое количество вспомогательных классов, облегчающих работу с рефлексией в scala

- **Type** - Содержит всю информацию о типе и соответствующий ему символ. С помощью Type можно получить все родительские классы, члены этого типа, как наследованные так и определенные непосредственно в данном типе. Так же Type позволяет сравнивать типы
- **Symbol** Все чему в scala можно дать имя имеет связанный символ. TypeSymbol, описывает определение тип. MethoSymbol - описание метода и т.д. Символы организованы в иерархию. Например символ, описывающий параметр метода, будет иметь родительский символ, описывающий сам метод
- **Trees** - это представление scala приложения в виде AST. Обычно tree неизменны, кроме нескольких полей, которые устанавливаются после typecheck фазы компилятора. Чаще всего Trees используют в макросах и в случаях применения метода **scala.reflect.api.Universe#reify**
- **Names** представляют имена термов и типов
- **Annotations** - аннотаций

Рефлексия и макросы

Scala reflection

Также scala-reflect предоставляет средства сохранить информацию о типах, которая теряется на этапе type erasure. Сделано это с помощью набора тегов

- **WeakTypeTag -** применим для сохранения информации о тайп параметрах и тайпалиасах, даже если они частично определены. Т.е. **weakTypeTag** сохранит информацию о типе **List[T]**
- **TypeTag** подходит для сохранения информации о конкретных типах. Попытка найти тайптег для типа **List[T]** завершится ошибкой компиляции
- **ClassTag -** редставляет информацию о типе, такой, какая она будет после typeerasure

Рефлексия и макросы

Scala reflection. Runtime Reflection
Теперь, когда мы познакомились с основными концепциями, давайте посмотрим как ими пользоваться.

Получение информации о типе и классе
Информация о классе, это единственная информация, которую можно получить не применяя scala-reflect. Сделать это можно с помощью метода **classOf[T]**

```
// получение информации о классе
val reflectExampleTraitCls = classOf[ReflectExampleTrait]
val reflectExampleCls = classOf[ReflectExample]
```

Для остальной работы в первую очередь импортируют вселенную и ее содержимое

```
import scala.reflect.runtime.{universe => ru}
import ru._
```

Рефлексия и макросы

Scala reflection. Runtime Reflection

```
// получение информации о типе
val reflectExampleTpe = typeOf[ReflectExampleTrait]
val reflectExTpe = typeOf[ReflectExample]
val declarations = reflectExTpe.decls
val baseClasses = reflectExTpe.baseClasses
assert(reflectExTpe <:< reflectExampleTpe)

// получение информации о типе и классе из инстанса
val exampleInst = new ReflectExample
val classSymbol = mirror.classSymbol(exampleInst.getClass)
val typeFromSymbol = classSymbol.asType.toType
assert(reflectExTpe =:= typeFromSymbol)
```

Рефлексия и макросы

Scala reflection. Runtime Reflection

Динамическое создание инстанса класса

```
// Динамическое создание инстанса класса
// Если попробовать создать инстанс трейта, будет ошибка
// val traitClassSymbol = mirror.classSymbol(reflectExampleTraitCls)
val traitClassSymbol = mirror.classSymbol(reflectExampleCls)
val classMirror = mirror.reflectClass(traitClassSymbol)
val constructor = traitClassSymbol.asType.toType.decl(ru.termNames.CONSTRUCTOR).asMethod
val reflectedConstructorMirror = classMirror.reflectConstructor(constructor)
val dynamicInst = reflectedConstructorMirror.apply()
```

Рефлексия и макросы

Scala reflection. Runtime Reflection.Тэги

Тэги создаются на этапе компиляции и содержат всю информацию о типе, которому принадлежат. Для того чтобы тег был создан, нужно “попросить” компилятор его создать. Это можно сделать следующими способами

- вызвав специальный метод в зависимости от типа тэга, **typeTag[T]**, **weakTypeTag[T]**, **classTag[T]**. При этом **T** должен быть реальным типом для всех тегов, кроме weakTypeTag
- передать его имплиситным параметром в метод, например так:
  def weakParamInfo[T](x: T)(implicit tag: WeakTypeTag[T])
- указать тег как ограничение контекста для типа параметра
  def patternMathWithTypeTag[T: TypeTag](t: T)

Тэги, по сути, являются обертками над инстансами **Type** и следовательно мы можем сделать все, что описано выше с типами. Кроме того, становится возможным писать паттерн мэтчинг не чувствительный type erasure.
Примеры работы с тегами **lectures.reflection.UsingScalaTags.scala**

Рефлексия и макросы

Scala reflection. Работа с АСД кода
Любой скала код возможно представить в виде абстрактного синтаксического дерева (АСД).
Узлами такого дерева являются наследники типа **Tree.** Далее приведен неполный список таких нод

- Подкласс **TermTree** имеет следующих наследников:
  - **Apply** представляют собой вызов методов
  - **New** - методы создания новых инстансов
  - **Literal -** применение литеральных значений в коде
- Подкласс **TypTree,** содержащий упоминания типов явно указанных в коде например так, **List[Int]**.
- Подкласс **SymTree** имеет несколько наследников.
  - **ClassDefs** - описание создания класса или трейта
  - **ValDef** - описание полей, параметров, переменных и т.д.
  - **Idents** - представление в ссылки на существующее описание, например меренная или метод

Например код

- **val x = 7** будет представлен в виде АСД, как **ValDef(Modifiers(), TermName("x"), TypeTree(), Literal(Constant(7)))**

Рефлексия и макросы

Scala reflection. Работа с АСД кода. Reify
Т.к строить вручную довольно трудно, есть несколько способов упростить эту задачу. Один из таких методов - это использовать метод **reify.** Он принимает scala выражение типа **T.** Результат **reify** представление кода, переданного выражения, обернутое в тип **Expr[T].**

```
val expr = reify[AnyRef]({
 case class RTest(i: Int)
 RTest(10)
})
```

Код, представленный одним **Expr[T]** может быть использован внутри описания выражения другого **Expr[U]**. Такое переиспользование называется сплайсинг кода.

```
val expr2 = reify ({
 class RTTestContainer(){
   expr.splice
 }
})
val code = show(expr2.tree)
```

Рефлексия и макросы

Scala reflection. Работа с АСД кода. Квазиквоты

Квазиквоты - это строковые интерполяторы вида **q”...”**, **tq”...”**, **pq”...”,** которые порождают **Tree** из переданных им строк. Здесь мы рассмотрим самый востребованный интерполятор **q,** который позволяет интерполировать любые выражения, определения и импорты.

```
val quoteExpression = q"""
   print("quasiquotes are awesome")
  """
val complicatedQuoteExpression =  q"""
    case class QQTest(i: Int)
     val qqi =  QQTest(10)
     print(qqi.i)
  """
showRaw(quoteExpression)
val tree = showRaw(complicatedQuoteExpression)
```

Если проанализировать **complicatedQuoteExpression,** мы увидим, что она представляет собой **Tree** достаточно сложной структуры, которую уже довольно сложно создать вручную.

Рефлексия и макросы

Scala reflection. Работа с АСД кода. Квазиквоты

```
Block(List(
 ClassDef(Modifiers(CASE), TypeName("QQTest"), List(),
   Template(List(
     Select(Ident(scala), TypeName("Product")),
     Select(Ident(scala), TypeName("Serializable"))
   ), noSelfType,
     List(ValDef(Modifiers(CASEACCESSOR | PARAMACCESSOR), TermName("i"), Ident(TypeName("Int")), EmptyTree),
       DefDef(Modifiers(),
         termNames.CONSTRUCTOR,
         List(),
         List(
           List(ValDef(Modifiers(PARAM | PARAMACCESSOR), TermName("i"), Ident(TypeName("Int")), EmptyTree))
         ),
         TypeTree(),
         Block(List(pendingSuperCall), Literal(Constant(())))
     )
   )),
 ValDef(Modifiers(), TermName("qqi"), TypeTree(), Apply(Ident(TermName("QQTest")), List(Literal(Constant(10))))),
 Apply(Ident(TermName("print")), List(Select(Ident(TermName("qqi")), TermName("i"))))
))
```

Рефлексия и макросы

Scala reflection. Работа с АСД кода. Квазиквоты

Как и в случае с reify, в квазиквоты можно встраивать инстансы **Tree,** полученные ранее

```
val tree = q"{val x = 10; x}"
val tree2 =q"print"
val treeCombined = q"$tree2($tree)"
val combinedResult = show(treeCombined)
// combinedResult будет выглядеть примерно так
print({
val x = 10;
x
})
```

Квазиквоты имеют метод upapply и могут использоваться для деконструкции деревьев на составные части

```
val q"new $t[..$_](...$pargs)" = reify(new ReflectExample[Int](1)(2)).tree
show(t) // Select(ScalaMacroExamples.this.ReflectExample)
show(pargs)// List(List(Literal(1)), List(Literal(2)))
```

$name - это именованная переменная, содержащая соответствующее дерево. **..$name** - это **List[Tree]** деревьев. **...$name** - это **List[List[Tree]]**

Рефлексия и макросы

Scala reflection. Задание
lectures.reflection.RuntimeReflectionSerializer.scala

Рефлексия и макросы

Scala reflection. Макросы
Макрос - это функция особой формы, которая, на этапе компиляции, позволяет анализировать и модифицировать код приложения. Описание метода-макроса ничем не отличается от обычного метода. Правая часть начинается с ключевого слова **macro** за которым следует вызов метода, реализующего логику макроса. Существует 2 различных стиля описания реализаций макросов

- **классический** Метод, располагается внутри **object** и принимает 2 набора параметров. Первый набор всегда один из возможных контекстов **whitebox.Context** или **blackbox.Context.** Второй набор параметров - это параметры имеющие тип **c.Tree** и совпадающие по количеству с параметрами основного метода**.** Тип **Tree** path-dependent относительно контекста, переданного первым параметром. Метод возвращает тип **c.Tree** или **c.Expr[T].** В примере ниже макрос принимает один параметр и всегда заменяет тело **scalaMacro** на число

```
def scalaMacro(prm: String): Any = macro ExampleMacro.generate

object ExampleMacro {
def generate(c: whitebox.Context)(prm: c.Tree):c.Tree = {
 import c.universe._
  reify(10).tree
}}
```

Рефлексия и макросы

Scala reflection. Макросы

- **bundle** В случае bundle, реализация находится в методе внутри класса. Метод принимает параметры имеющие тип **c.Tree** и совпадающие по форме и количеству с параметрами основного метода. Возвращаемый тип также **c.Tree** или **c.Expr[T].** Класс, содержащий реализацию должен обладать единственным публичным конструктором с параметром типа **whitebox.Context или blackbox.Context.** Реализация макроса в форме бандла предпочтительнее т.к. позволяет импортировать содержимое контекста один раз и не передавать его в качестве параметра во все внутренние методы, входящие в реализацию макроса. Ниже макрос из предыдущего примера,

```
def bundledMacro(prm: String): Any = macro BundledMacroExample.generateMore

class BundledMacroExample(val c: whitebox.Context) {
 import c.universe._
 def generateMore(prm: Tree): Expr[Int] = {
   reify(10)
 }}
```

Рефлексия и макросы

Scala reflection. Макросы
Имплементация макроса, должна быть скомпилирована отдельно от кода, в котором применяется. Поэтому макросы удобно размещать в отдельном билде или собирать в библиотеку.
Методы, содержащие макросы, являются, по сути, обычными методами. Они могут принимать тайп параметры, иметь несколько наборов параметров, включая имплиситные и оперировать тайп и класс тегами. Ниже представлен макрос, который создает краткое описание, переданного типа на этапе компиляции

lectures.reflection.DescriberMacro.scala

У разработчиков, которые только начинают знакомится с макросами часто возникает вопрос, как вернуть содержимое переменной вычисленной в макросе в виде части сгенерированного дерева. В примере выше макрос возвращает **c.Expr[String]**, где сгенерированный код представляет собой содержимое строковой переменной **res.** Чтобы поместить содержимое переменной в дерево, которое можно вернуть их макроса, можно воспользоваться 2-я подходами

- сконструировать дерево вручную: **c.Expr[String](Literal(Constant(res)))**
- поместить переменную в квазиквоту: **c.Expr[String](q"$res"),** если для данного типа имплиситно присутствует **Liftable[T].** Для строки**, Liftable** существует из коробки

Рефлексия и макросы

Scala reflection. Макросы. Контексты
Поведение макроса сильно зависит от того, какой контекст был в него передан. Существует две разновидности контекстов: **blackbox.Context** и **whitebox.Context**
Blackbox обязывает макрос строго следовать своему возвращаемому типу. Этот тип описывается сигнатурой метода, содержащего макрос. Кроме того, все типы, участвующие в вызове макроса, должны быть вычислены до раскрытия макроса (т.е. подстановки на место вызова макроса, вычисленного им AST)
Whitebox не следует сигнатуре метода. Определяющим для него является тип дерева, получившегося после раскрытия. **Whitebox** макрос позволяет начать раскрытие, даже если компилятор не смог вычислить все типы, участвующие в сигнатуре метода, содержащего макрос. После раскрытия макроса, компилятор попытается вычислить оставшиеся типы из типа получившегося дерева. Пример в объекте **ExampleMacro**, макрос **generate** и его применение в тесте **ScalaMacroExampleTest.**
Благодаря большей гибкости **whitebox** макросы позволяют реализовать несколько интересных техник, о которых можно подробнее прочесть в документации

- fundep materialization и пулреквест в скалу, который тоже содержит неплохое описание сути техники
- динамическое вычисление наличия подходящего макроса

Рефлексия и макросы

Scala reflection. Макросы.
Вне зависимости от контекста можно сделать так, что бы тип возвращаемого дерева зависел от параметров переданных в макрос. Таким образом макросы так же могут быть полиморфными, как обычные методы. **Пример ScalaMacroExamples метод scalaMacroT**

Как и обычные методы, макросы могут быть имплиситными. При этом в зависимости от контекста поведение таких макросов немного отличается.

- **blackbox** выбросит исключение, если в нем будет вызван метод **c.abort**
- **whitebox** завершится без ошибки. Компилятор попробует подобрать другой подходящий макрос.

Рефлексия и макросы

Scala reflection. Макросы. Отладка
Отлаживать макросы довольно проблематично, т.к. выполнение кода макроса происходит на этапе компиляции. Соответственно брейкпоинты и дебаг недоступны. Если код макроса не получается протестировать вне рамок макроса, можно воспользоваться вспомогательными функциями, которые немного упрощают задачу:

- **show** - метод превращающий Tree в строку представляющую scala код, который может быть получен из этого дерева
- **showRaw -** распечатывает дерево в терминах AST
- **context.info -** выводит произвольное сообщение в консоль на этапе раскрытия макроса
- **context.abort -** завершает раскрытие макроса с ошибкой

задание: lectures.reflection.MacroSerializer.scala