Metaprogramming in Scala

CERN lightnight talk

Florian Cassayre

7th November 2025

Metaprogramming

A set of techniques in which programs have the ability to *analyze*, *generate*, or *transform* other programs—including themselves.

Goals:

- Increase abstraction
- Reduce boilerplate
- Improve runtime performance

Some examples (of varying capabilities):

- **C++**: templates
- **JavaScript**: `eval`, proxy objects, reflection
- **Java**: reflection, annotation processors

Metaprogramming in Scala 3

- **Runtime reflection**: access the high level structure
- **Inlining**: a modifier to guarantee inline at point of use
- **Macros**: convert program code to data, and vice versa
- **Runtime multi-staging**: compile code at runtime
- **Compile-time operations**: e.g. convert types to values

Scala

- Invented in 2004 at EPFL
- Runs on the JVM
- Statically typed, multi-paradigm and versatile
- Focus on type safety and expressiveness
- Complete revamp of the language in version 3

Function inlining: recursion

Replace a function **call site** with the **body** of the called function.

```
inline def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n - 1)
```

```
val f = factorial(3)
```

Function inlining: recursion

Replace a function **call site** with the **body** of the called function.

```
inline def factorial(n: Int): Int =
  if n == 0 then 1
  else n * factorial(n - 1)
```

```
val f = factorial(3)
```

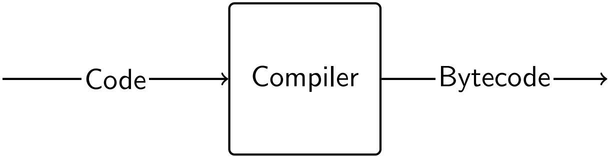
$$\Downarrow$$

```
// 3 * (2 * (1 * (1)))
val f = 6
```

(the compiler also performs a **constant folding** optimization)

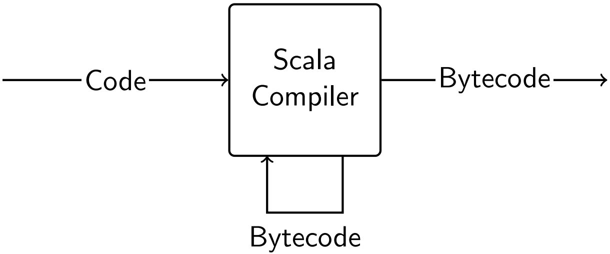
Scala compiler

A compiler takes **code** and produces **machine code** (or **bytecode** in the context of the JVM).



Scala compiler

A compiler takes **code** and produces **machine code** (or **bytecode** in the context of the JVM).



When the Scala compiler encounters a macro, it **compiles** it, **executes** the compiled bytecode, and then proceeds to compiles the returned **AST**.

Macros: quotation and splicing

- **Quotation**: converting **code** into an **AST**
- **Splicing**: converting an **AST** back into **code**

$$\texttt{T} \; \underset{\texttt{\$\{ * \}}}{\overset{\texttt{'\{ * \}}}{\rightleftarrows}} \; \texttt{Expr[T]}$$

Assertion macro

```
inline def assert(inline cond: Boolean): Unit =
  ${ assertImpl('cond) }

def assertImpl(cond: Expr[Boolean])(using Quotes): Expr[Unit] =
  '{ if !$cond then throw AssertionError(${Expr(cond.show)}) }
```

Assertion macro

```
inline def assert(inline cond: Boolean): Unit =
  ${ assertImpl('cond) }

def assertImpl(cond: Expr[Boolean])(using Quotes): Expr[Unit] =
  '{ if !$cond then throw AssertionError(${Expr(cond.show)}) }
```

```
assert("foo".equals("bar"))
```

Assertion macro

```
inline def assert(inline cond: Boolean): Unit =
  ${ assertImpl('cond) }

def assertImpl(cond: Expr[Boolean])(using Quotes): Expr[Unit] =
  '{ if !$cond then throw AssertionError(${Expr(cond.show)}) }
```

```
assert("foo".equals("bar"))
```

$$\Downarrow$$

```
if !"foo".equals("bar") then
  throw AssertionError("\"foo\".equals(\"bar\")")
```

Parsing string contexts

String contexts are syntactic sugars that support splicing (= injection of parameters):

```
url"http://example.com"
url"not_a_url" // Compilation error
```

Parsing string contexts

String contexts are syntactic sugars that support splicing (= injection of parameters):

```
url"http://example.com"
url"not_a_url" // Compilation error
```

```
extension (inline sc: StringContext)
  inline def url(inline args: Any*): URL =
    ${ urlImpl('sc, 'args) }

def urlImpl(scExpr: Expr[StringContext], argsExpr:
    Expr[Seq[Any]])(using Quotes): Expr[URL] =
  val rawUrl = scExpr.valueOrAbort.parts.mkString
  Try(URL(rawUrl)) match
    case Success(_) => '{ URL(${ Expr(rawUrl) }) }
    case Failure(_) => quotes.reflect.report
      .errorAndAbort(s"Invalid URL: '$rawUrl'")
```

Memoization macro

```scala
@memo
def fibonacci(n: Int): Int =
  if n < 2 then n else fibonacci(n - 2) + fibonacci(n - 1)
```

$\Downarrow$

```scala
private val fibonacci: mutable.HashMap[Tuple1[Int], Int] =
  mutable.HashMap()

@memo
def fibonacci(n: Int): Int =
  fibonacci.getOrElseUpdate(
    Tuple1(n),
    if n < 2 then n else fibonacci(n - 2) + fibonacci(n - 1)
  )
```

Runtime multi-stage programming

The runtime can be bundled with the Scala compiler itself. This can be used to implement a typesafe JIT compiler:

```
given Compiler = Compiler.make(getClass.getClassLoader)

staging.run:
  '{ println("Hello world!") }
```

Scala libraries

A non-exhaustive list of libraries exploiting these language features:

- **sbt**: the standard Scala compiler
- **scalatest/scalactic**: debug-friendly assertions
- **refined**: data constraints represented as types
- **shapeless**: a framework that implements *generic programming*
- **quill**: DSL for type-safe SQL queries
- **magnolia**: automatic *typeclass derivation* for ADTs

FlorianCassayre/ITLT27-scala-metaprogramming