# Runtime Multi-Stage Programming



The framework expresses at the same time compile-time metaprogramming and multi-stage programming. We can think of compile-time metaprogramming as a two stage compilation process: one that we write the code in top-level splices, that will be used for code generation (macros) and one that will perform all necessary evaluations at compile-time and an object program that we will run as usual. What if we could synthesize code at run-time and offer one extra stage to the programmer? Then we can have a value of type `Expr[T]` at run-time that we can essentially treat as a typed-syntax tree that we can either *show* as a string (pretty-print) or compile and run. If the number of quotes exceeds the number of splices by more than one (effectively handling at run-time values of type `Expr[Expr[T]]`, `Expr[Expr[Expr[T]]]`, ...) then we talk about Multi-Stage Programming.

The motivation behind this *paradigm* is to let runtime information affect or guide code-generation.

Intuition: The phase in which code is run is determined by the difference between the number of splice scopes and quote scopes in which it is embedded.

- If there are more splices than quotes, the code is run at compile-time i.e. as a macro. In the general case, this means running an interpreter that evaluates the code, which is represented as a typed abstract syntax tree. The interpreter can fall back to reflective calls when evaluating an application of a previously compiled method. If the splice excess is more than one, it would mean that a macro's implementation code (as opposed to the code it expands to) invokes other macros. If macros are realized by interpretation, this would lead to towers of interpreters, where the first interpreter would itself interpret an interpreter code that possibly interprets another interpreter and so on.

- If the number of splices equals the number of quotes, the code is compiled and run as usual.
- If the number of quotes exceeds the number of splices, the code is staged. That is, it produces a typed abstract syntax tree or type structure at run-time. A quote excess of more than one corresponds to multi-staged programming.

Providing an interpreter for the full language is quite difficult, and it is even more difficult to make that interpreter run efficiently. So we currently impose the following restrictions on the use of splices.

1. A top-level splice must appear in an inline method (turning that method into a macro)
2. The splice must call a previously compiled method passing quoted arguments, constant arguments or inline arguments.
3. Splices inside splices (but no intervening quotes) are not allowed.

# API

The framework as discussed so far allows code to be staged, i.e. be prepared to be executed at a later stage. To run that code, there is another method in class `Expr` called `run`. Note that `$` and `run` both map from `Expr[T]` to `T` but only `$` is subject to the PCP, whereas `run` is just a normal method. `scala.quoted.staging.run` provides a `Quotes` that can be used to show the expression in its scope. On the other hand `scala.quoted.staging.withQuotes` provides a `Quotes` without evaluating the expression.

```
package scala.quoted.staging

def run[T](expr: Quotes ?=> Expr[T])(using Compiler): T = ...

def withQuotes[T](thunk: Quotes ?=> T)(using Compiler): T = ...
```

# Create a new Scala 3 project with staging enabled

```
sbt new scala/scala3-staging.g8
```

From `scala/scala3-staging.g8`.

It will create a project with the necessary dependencies and some examples.

In case you prefer to create the project on your own, make sure to define the following dependency in your `build.sbt` build definition

```
libraryDependencies += "org.scala-lang" %% "scala3-staging" % scalaVersion.valu
```

and in case you use `scalac` / `scala` directly, then use the `-with-compiler` flag for both:

```
scalac -with-compiler -d out Test.scala
scala -with-compiler -classpath out Test
```

# Example

Now take exactly the same example as in Macros. Assume that we do not want to pass an array statically but generate code at run-time and pass the value, also at run-time. Note, how we make a future-stage function of type `Expr[Array[Int] ⇒ Int]` in line 6 below. Using `staging.run { ... }` we can evaluate an expression at runtime. Within the scope of `staging.run` we can also invoke `show` on an expression to get a source-like representation of the expression.

```
import scala.quoted.*

// make available the necessary compiler for runtime code generation
given staging.Compiler = staging.Compiler.make(getClass.getClassLoader)

val f: Array[Int] => Int = staging.run {
  val stagedSum: Expr[Array[Int] => Int] =
    '{ (arr: Array[Int]) => ${sum('arr)}}
  println(stagedSum.show) // Prints "(arr: Array[Int]) => { var sum = 0; ... }"
  stagedSum
}

f.apply(Array(1, 2, 3)) // Returns 6
```

‹ Macro... Reflect... ›