Parser Macros for Scala

Martin Duhem

School of Computer and Communication Sciences
Semester Project

Supervisor
Eugene Burmako
EPFL / LAMP

Supervisor
Prof. Martin Odersky
EPFL / LAMP

June 2015

Contents

1 Introduction

The Scala programming language already supports, in its official distribution or through the use of external plugins, several kinds of macros.

Since their introduction, macros have been used in many innovative ways by developers for various tasks such as reducing the amount of boilerplate in their source code and more particularly defining new domain specific languages embedded into Scala. Examples of projects that use macros are Slick [1], a framework that facilitate database querying in Scala, or Shapeless [2], a generic programming library for Scala.

However, even if macros have brought a lot of value for DSL implementors, they still do not expose a mechanism that would allow developers to specify their own syntax, which may not follow Scala's grammar.

The compiler plugin that we present introduces a flavor of macros, called parser macros, which attempts to provide developers with a mean to define macros whose arguments can be written in arbitrary Scala-like syntax. Along with the plugin, we also present the facilities that we implemented in order to make it more comfortable to write parser macros.

What are the new possibilities that this plugin offers? Is it powerful enough to emulate basic constructs from the Scala programming language? What are its drawbacks, and how does it interact with the other, already existing macro flavors? We will bring answers to these questions in the present report.

2 Presentation of parser macros

Parser macros offer to developers a way to define macros that will accept tokens as arguments, as opposed to usual Scala macros which take trees as arguments. If a macro accepts trees as arguments, it means that its arguments must be parseable according to the Scala grammar [3]. For `def macros` this is even more restrictive, since the arguments must also be type-correct.

However, in certain cases, we may want to give to our macros arguments that are not even valid Scala expression from the point of view of Scala's syntax analyzer. In such situations, parser macros allow the developers to write a macro implementation that will be in charge of making sense of the sequence of tokens that it receives as argument, and do something with it. This implementation will then be callable in classic Scala source code, and it will accept arguments that are not valid Scala code.

2.1 What are tokens?

Tokens are atomic elements of source code. They are created from an input stream and correspond to pieces of it. For instance, the character stream `for { x <- lst } yield x + 1` corresponds to the following sequence of tokens: `for, {, x, <-, lst, }, yield, x, +, 1` (tokens representing whitespaces have been ommitted).

We argue that tokens are at the right level of abstraction for parsing: because each token represents a subpart of the source, they are easier to handle than strings. Moreover, they are more precise than strings, because they carry all the information of their input stream (exact position in the stream, origin of the stream, etc.). Finally, a higher-level structure (such as `AST`s for instance) would require us to be able to make sense of the input stream, which would in turn restrict the scope of parseable streams.

```
object Hello extends App {
  Lib.enum#(WeekDays)#(Mon (...) Sun)
  import WeekDays._
  def todaysMood(day: Value) = day match
    {
    case Mon => "Mondays are bad"
    case Fri => "Fridays are better"
    case Sat | Sun => "Weekends are best"
  }

  println(todaysMood(WeekDays.Fri))
}
```

Listing 1: An application of Listing 3

```
object WeekDays {
  sealed trait Value
  case object Mon extends Value
  (...)
  case object Sun extends Value
}
```

Listing 2: An expansion of `enum`.

2.2 Writing a simple parser macro

First, let us present a simple example of parser macro that will emulate enumerations. The goal of this parser macro is to be used to insert the definition of a new object which could be used as an enumeration, while bringing more value than standard Scala enumerations by enabling exhaustivity checks in pattern matching. Such a parser macro is presented in Listing 3, while its application can be found in Listing 1.

The second line of Listing 3 is the signature of the parser macro. It looks just like the signature of a classic Scala method, except that it has the keyword `macro` at the beginning of its body. This parser macro takes two arguments, both of type `scala.meta.Tokens` and returns a `scala.meta.Tree`. This signature conforms to the expected signature of parser macros, whose properties must match the conditions described in subsection 3.4.

This implementation will then create a name for the enumeration from the first argument of the parser macro, and synthetize `case objects` that will represent the values of this enumeration. Finally, an `object` that represents the whole enumeration is created. This object holds the different values of the enumeration.

The application of a parser macro also has the feel of a normal Scala function application, except that it has a special character `'#'` just before

```
object Lib {
  def enum(name: Tokens, values: Tokens): Tree = macro {
    def TermName(name: String) = name.parse[Term.Name]
    val enumName = TermName(name(1).code)
    val enumValues =
      values filterNot (_.isTrivia) map { t =>
        val name = TermName(t.code)
        q"case object $name extends Value"
      }

    q"""object $enumName {
          sealed trait Value
          ..${enumValues.toList}
        }"""
  }
}
```

Listing 3: A simple parser macro that creates enumerations

the opening parenthesis, as shown in Listing 1. The reason for this special token is explained in subsection 3.1.

During the compilation of a parser macro application, the parser macro implementation is going to be invoked and will receive the sequence of tokens corresponding to the arguments that it is given. The expansion of a parser macro gives rise to a tree that will be spliced in the original program, in place of the parser macro application.

The result from the expansion of the parser macro shown in Listing 1 is shown in Listing 2.

Despite being extremely simple, this implementation offers advantages over Scala's implementation of enumerations. First, it allows a syntax that is shorter and less puzzling than Scala's. Secondly and more importantly, it enables the compiler to make exhaustivity checks in pattern matching for values of the enumeration. For instance, compiling the code presented in Listing 1 will produce a warning about exhaustivity check, saying that the values `Tue`, `Wed` and `Thu` are not covered by pattern matching.

```
@parsermacro("Hello, world!") val willDisappear = 0
```

Listing 4: Emulating parser macros with macro annotations

2.3 Related work

One major argument for parser macros is that they accept as input a sequence of tokens that may not represent valid Scala code. However, this behavior could easily be emulated using macro annotations.

For instance, one may create a macro annotation that would receive as argument a `String`, tokenize it, and finally reason about this sequence of tokens. An example of what this kind of macro would look like is shown in Listing 4.

However, this solution does not give us complete satisfaction for several reasons.

First, the arguments have to be given as a string, which means that one would have to enclose them between double quotes and escape some characters that they may want to use as arguments of a parser macro. Moreover, this string will most likely not contain the amount of metadata that token bring, as discussed ealier.

Secondly, to use macro annotations, users would have to insert a dummy declaration that will host the macro annotation. This increases the amount of code required to make use of parser macros and obfuscates the source: why is there a value declaration which we know will be removed during compilation?

Thirdly, the definition of a macro annotation requires more code than the definition of a parser macro. Where for a parser macro you only need one method in a static object, macro annotations require one class definition that represents the type of the annotation and one method in a static object.

Finally, because they have to be attached to a declaration, macro annotations do not expand in expression position, which means that they are not able to produce a simple value that is immediately usable, like in our example in Listing 1.

3 Implementation of parser macros

The Scala compiler doesn't provide any standard way to emulate something like our proposition of parser macros, which means that we need a way to inject the behavior that we desire in the Scala compiler.

Fortunately, the Scala compiler has the required infrastructure to host plugins. The Scala compiler `API` exposes *hooks* which allow compiler plugins to participate in several phases of the compilation process.

First, our compiler plugin must intervene during parsing in order to make Scala's syntax analyzer accept parser macro applications and perform rewritings in certain cases.

Then, the plugin also requires to act during the naming and typing phases of a Scala program, in order to generate a synthetic implementation of a parser macro and finally expand parser macro applications.

3.1 Modifying Scala's grammar

The code that is shown in Listing 1 cannot be parsed by a Scala compiler without some adaptations. These adaptations are required to make the syntax analyzer understand that when it encounters a method call followed by the special character `'#'`, then this method call must be treated as a parser macro application.

Moreover, because parser macros have the ability to expand into definitions, it makes sense to allow them to expand into top level position. Again, this change implies modifying the syntax analyzer, because Scala's grammar do not allow function application in top level position.

The changes made to Scala's grammar are shown in Grammar 1. The original definition of Scala's grammar can be found in [3].

As the grammar shows, parser macro applications resemble normal function applications a lot, but are different in the way they are given arguments. When multiple parameters are given to a function, they must be separated by commas, where for parser macro applications each parameter must be enclosed in a different set of parentheses which is prefixed by `'#'`.

Requiring this special character at the beginning of parser macro appli-

$$
\begin{array}{lll}
\langle \textit{TopStat} \rangle & ::= & \text{Unchanged} \\
& | & \langle \textit{PMacro} \rangle \\
\langle \textit{PMacro} \rangle & ::= & \langle \textit{QualId} \rangle \ (\text{`.'} \ \langle \textit{PMacro} \rangle) \mid \langle \textit{PMacroApp} \rangle \\
\langle \textit{PMacroApp} \rangle & ::= & \text{`\#'} \ (\text{`(')} \ \langle \textit{Anything} \rangle \ \text{`)'} \mid \text{`\{'} \ \langle \textit{Anything} \rangle \ \text{`\}'}) \ [\langle \textit{PMacroApp} \rangle] \\
\langle \textit{SimpleExpr1} \rangle & ::= & \text{Unchanged} \\
& | & \langle \textit{PMacroApp} \rangle \\
\langle \textit{Anything} \rangle & ::= & \text{Any sequence of characters}
\end{array}
$$

Grammar 1: Modifications to Scala's grammar

cations is a limitation of parser macros that we are aware of, but which announces itself as hard to fix. In the current implementation, this character is used to distinguish parser macro applications from normal function applications. This special token will tell the scala parser that the next token should be an opening brace or parenthesis, and that it should not try to make sense of the input until it has seen the matching closing brace or parenthesis.

If we were to remove this limitation, we could decide that a function application becomes a parser macro application if we are unable to parse its arguments, and finally verify during typing that the application is indeed a parser macro application. However, this solution would not quite work for parser macros: If the arguments of a parser macro application parse, then we won't be able to get back the exact character stream that produced the tree because of the desugarings that `scalac` performs during parsing, as shown in Listing 5.

The implementation that we propose will parse a parser macro application such as `Provider.impl#(foo)#(bar)` as the tree corresponding simply to `Provider.impl`, and will attach to it the arguments of this parser macro, which are, in our case, `List("foo", "bar")`. Please note that, at this point, the arguments are represented by a list of strings.

We will then be free to extract this attachment when we need these arguments.

```
scala> import scala.reflect.runtime.universe._
scala> q"a map b"
res0: Tree = a.map(b)
scala> q"for (x <- List(1, 2, 3)) yield x"
res1: Tree = List(1, 2, 3).map(((x) => x))
```

Listing 5: Informations are lost during the desugarings performed by `scalac`

```
@hello
object Foo extends App {
  sayHello
}
```

Listing 6: `sayHello` is not defined before macro expansion.

3.2 Rewriting parser macro applications

To make parser macros more useful, we wanted them to be able to expand into new definitions. However, writing a new macro engine that would be able to expand macro applications into new publicly visible definitions is a very complex problem, for several reasons.

The expansion of a macro that is able to introduce new publicly visible definitions must take place before the program has been typed, because parts of it may not typecheck without the definitions that may be introduced by the macro expansion. Because definitions are statements according to Scala's grammar, we consider that a parser macro application is able to introduce new definitions if it appears in statement position. An example of such code is shown in Listing 6, where the macro annotation `@hello` will add a new method `sayHello` to object `Foo`.

Some parts of the program must obviously be typed to be able to perform the macro expansion (for instance, the selection of the macro annotation `@hello` must be typed in Listing 6).

Fortunately, there already exists a compiler plugin, Macro Paradise [4], that is able to expand macro annotations into definitions. To give this ability to parser macros as well, we decided to rewrite macro applications in state-

```
class Foo {
  Lib.enum#(X)#(Y Z)
}
```

```
class Foo {
  @ParserMacroExpansion(Lib.enum)
  object TemporaryObject {
    val tokens = List("X", "Y Z")
  }
}
```

Listing 7: Rewriting parser macro applications to macro annotated objects

ment position to dummy definitions with a macro annotation, as shown in Listing 7.

3.3 Generating synthetic implementations

During the compilation of a macro client, the expansion process of parser macro requires to use Java reflection to invoke the macro implementation. The result of the call to the macro implementation will then be inlined in the macro client and completely replace the original parser macro application.

However, the methods that are marked with the `macro` keyword do not appear as normal methods in the `classfile`s that result from the compilation of a macro provider, and are therefore not visible to Java reflection.

To overcome this problem, our solution is to generate on the fly a new `private` method in the `class` of the macro provider. The body of this method is the code of the original macro implementation. Because this synthetic method is no longer marked with the `macro` keyword, it will be visible to the eyes of Java reflection, but the original implementation will still appear to Scala clients.

To introduce this synthetic method, our plugin needs to inspect the declaration of every new method in the symbol. Fortunately, this can be done in a compiler plugin by overriding the hook `pluginsEnterStats`. Each time that the compiler encounters a statement (a method definition is a statement), then the Scala compiler will call `pluginsEnterStats` for every plugin that is enabled until one of them produces a value for this particular statement. If none of them produce a result, then the compiler will treat the node as it

would in the absence of plugins.

In the case of the parser macro compiler plugin, we are only interested at this phase in nodes that represent method definitions with some precise properties.

- The method must have the `macro` keyword;
- It must not be `implicit`;
- It must have exactly one parameter list; and
- It cannot have `implicit`, by-name or default parameters.

Obviously, this is not the complete list of conditions that must hold true for a parser macro to be considered as valid. In particular, we have currently made no assumptions about the types of the parameters of the parser macro, nor about its return type. These verifications will be made at a later stage, and cannot be performed directly at this point, because the trees that we operate on have not been typed yet.

3.4 Typechecking parser macro implementations

At a later point in the compilation pipeline, the Scala compiler will start typechecking the parser macro implementation. Once again, the Scala compiler exposes hooks that allow plugin writers to inject some custom logic during this phase by overriding the method `pluginsTypedMacroBody`.

This allowed us to perform some additional checks to verify that a given macro implementation is actually a *correct* parser macro implementation.

Given a potential parser macro implementation (that is, an implementation that satisfies the conditions explained in subsection 3.3), we must still perform the following verifications:

- Every formal parameter of the parser macro must be of type `scala.meta.Tokens`, or a supertype;
- The body of the macro implementation must typecheck to a subtype of `scala.meta.Tree`;

- The parser macro implementation must belong to a static object.

A macro implementation that satisfies all these conditions and those of subsection 3.3 is then considered a valid parser macro implementation, and macro clients should therefore be able to use it.

3.5 Signing parser macro implementations

If these checks succeed, that is, if the macro is indeed a valid parser macro, then we also attach a signature to the macro implementation. This signature will be used during the expansion of a macro application to get informations about the macro implementation, or as expected in the future, to leverage the need for separate compilation of macro providers and macro clients.

This signature takes the shape of an annotation that is added to the macro implementation.

This annotation will tell us what method to invoke to perform the expansion of the parser macro. To store this information, we decided to put in the annotation the whole node corresponding to the definition of the synthesized macro implementation.

This technique has the advantage of working using today's technology by simply extracting the name of the desired implementation from this annotation, while still being ready for tomorrow where we would like to be able to compile macro providers and expand their applications in the same compilation run. Unfortunately, the resulting class files are slightly larger than they need to be because of the potentially large annotation.

3.6 Expanding parser macro applications

There are actually two implementations of the expansion of parser macros in our project, one for parser macro applications that have been rewritten to macro annotated definitions, and one for those that have not. These two implementations are very similar, and only differ in the way the arguments of the parser macro application are extracted.

For non-rewritten parser macro applications, our compiler plugin is completely in charge of the expansion. To be able to take back control from the

Scala compiler when it comes to the expansion of a parser macro, we can use another hook, `pluginsMacroExpand`, which will allow us to verify that the symbol under expansion is a parser macro and prepare everything for the expansion.

In the case of parser macro applications that have been rewritten to a macro annotated definition, the control is given to Macro Paradise. This plugin will execute our implementation of a custom macro annotation, which will in turn verify that the arguments of the macro annotation correspond to a valid parser macro implementation and extract the arguments of the application, which are stored in `TemporaryObject.tokens`, according to the rewriting shown in Listing 3.2.

Thanks to the signature that we attached to the macro implementations, one can easily verify that a given symbol is a parser macro implementation or not, by simply looking up the annotations attached to this symbol. If the symbol under inspection is a valid parser macro implementation then it will have the expected signature.

The situation is slightly more complicated for parser rewritten applications, because the parameters of macro annotations are untyped. This means that we will have to typecheck them in order to be able to resolve the parser macro implementation.

After the parser macro implementation has been successfully resolved, we have to prepare the arguments that will be used during the expansion. In our case, preparing the arguments means tokenizing the parameters that we have extracted from the macro application. This step is required because the arguments that we attached to the application in subsection 3.1 are strings.

Once we have resolved the implementation and the arguments are ready, we can reflectively invoke the parser macro implementation. Because this implementation has been validated by our engine, we can safely assume that the result of this invocation will be a subtype of `scala.meta.Tree`.

The last step is to convert the `scala.meta` tree to a `scala.reflect` tree. At the time of writing, the facility that will be in charge of doing this in `scala.meta` is not yet ready, therefore we decided to transform the tree to its string representation, and finally parse it back to a `scala.reflect` tree.

This converted tree represent the result of the parser macro expansion. This is the output that is finally given back to the Scala compiler to be spliced in the final program.

3.7 Limitations of parser macros

Separate compilation of providers and clients As it is the case for the vanilla macros included in the official distribution of the Scala programming language, the macro providers and their clients cannot be compiled in the same compilation run.

Because parser macros are compiled using globally the same scheme, they suffer from the same limitation. However, we are planning on relaxing this limitation in the future.

As explained in subsection 3.5, the complete tree that represents the macro implementation is not lost and could be used to perform the macro expansion.

The stubs of an interpreter for `scala.meta` exist, but the interpreter is not usable at the moment. However, when this interpreter is finished, we are confident that we will be able to lift this limitation and perform the compilation and expansion of parser macros in the same compilation run.

Cannot introduce top level definitions The current implementation of parser macros cannot expand into top level definitions. Remember that applications of parser macros that may introduce new definitions are rewritten to macro annotated definitions, as shown in Listing 7. Unfortunately, Macro Paradise requires that the expansion of a macro annotated top-level object cannot change the name of the object. This limitation means that the expansion of a top-level parser macro application should *always* yield an `object` name `TemporaryObject`.

The workaround to this limitation is to expand parser macro within another object, and then import the content of this object.

4 Token quasiquotes

Quasiquotes [5] are a very practical notation that allow users to construct and deconstruct trees using a concise and expressive syntax.

Unlike usual Scala macros which take trees as arguments, parser macros take sequences of tokens. In order to make it easy and enjoyable to write parser macros, we wanted to have a syntax that would allow parser macro writers to easily construct and deconstruct sequences of tokens.

Our proposition to perform this task is a new string interpolator whose behavior resembles that of the tree quasiquotes, but which operates on tokens. The interpolator can be enabled by using `toks""`.

4.1 Using the interpolator

This quasiquotes can be used, for instance, to construct new sequences of tokens:

```
scala> import scala.meta._ ; import scala.meta.dialects.Scala211
scala> toks"Hello, world!"
res0: Tokens = Tokens(Hello (0..5), , (5..6),   (6..7), world (7..12), !
    (12..13))
```

One can use the token quasiquotes to augment a sequence of `Token`s using one or more tokens:

```
scala> val lorem = toks"Lorem".head
scala> val dolorSitAmet = toks"dolor sit amet"
scala> toks"$lorem ipsum $dolorSitAmet"
res1: Tokens = Tokens(Lorem (0..5),   (0..1), ipsum (1..6),   (6..7),
    dolor (0..5),   (5..6), sit (6..9),   (9..10), amet (10..14))
```

It is also possible to deconstruct a sequence of tokens using the quasiquotes, which allows its use in pattern matching:

```
scala> val orig = toks"Quasiquotes allow construction and much more!"
scala> val toks"Quasiquotes allow $what and ..$rest!" = orig
what: Token = construction (18..30)
rest: Tokens = Tokens(much (35..39),   (39..40), more (40..44))
```

4.2 Limitations of the token quasiquotes

Rigid matching Unfortunately, it seems that the token quasiquotes, in its current implementation at least, does not provide the right level of abstraction to its users, because it is inflexible regarding matching. Therefore, it makes it hard to use it to parse the input of parser macros:

```
scala> val toks"hello, world!" = toks"hello,  world!" // two spaces
scala.MatchError: Synthetic(Vector(hello (0..5), , (5..6),   (6..7),
    (7..8), ...
```

The snippet of code above will fail at runtime because the two sequences of tokens differ by a single space. One solution to fix this problem could be to define multiple interpolators for the token quasiquotes. Each of these quasiquotes would treat whitespaces differently. Macro writers could then choose how strict they want the matching to be.

Usage within parser macros Because of an unresolved issue, most likely in the implementation of our compiler plugin, it is not possible to use token quasiquotes inside parser macros defined outside of the Scala `REPL`, because the `classfile`s that are produced by parser macros that use the token quasiquotes are corrupted.

5 Conclusion

We have presented a new compiler plugin that implements the support for parser macros in the Scala compiler. This compiler plugin takes advantage of the facilities exposed by `scala.meta` to provide `API`s to macro writers. Moreover, we have proven the immediate usefulness of this plugin by giving one really short example of parser macro, whose goal was to introduce enumerations in the Scala programming language.

In the course of our experimentations with this plugin, we have also written other examples of parser macros. One interesting experiment was to re-implement Scala's `for`-comprehension and performing the desugarings that happen during the parsing of Scala code [6]. Interestingly enough, our implementation seems to be a bit shorter than the corresponding code in the Scala compiler.

This new flavor of macro opens new possibilities and allows developers to write their source code using a customized and reusable syntax, in order to increase the expressiveness of their source code.

We are confident that this plugin can bring real value to Scala developers and allow them to come up with new innovative syntaxes.

References

[1] Slick, Functional Relational Mapping for Scala. http://slick.typesafe.com, 2015.

[2] Shapeless, Generic programming for Scala. https://github.com/milessabin/shapeless, 2015.

[3] Scala Language Specification, Syntax summary. http://www.scala-lang.org/files/archive/spec/2.11/13-syntax-summary.html, 2015.

[4] Macro Paradise Plugin. https://github.com/scalamacros/paradise, 2015.

[5] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, 2013.

[6] Re-implementing `for`-loops using parser macros. https://github.com/Duhemm/parsermacro-example, 2015.