Table of Contents for Programming Scala, 2nd Edition

Chapter 20. Domain-Specific Languages in Scala

A *domain-specific language* (DSL) is a programming language that mimics the terms, idioms, and expressions used among experts in the targeted domain. Code written in a DSL reads like structured prose for the domain. Ideally, a domain expert with little experience in programming can read, understand, and validate this code, if not also write code in the DSL.

We will just scratch the surface of this large topic and Scala's support for it. For more in-depth coverage, see the DSL references in Appendix A.

Well-crafted DSLs offer several benefits:

Encapsulation
: A DSL hides implementation details and exposes only those abstractions relevant to the domain.

Productivity
: Because implementation details are encapsulated, a DSL optimizes the effort required to write or modify code for application features.

Communication
: A DSL helps developers understand the domain and domain experts to verify that the implementation meets the requirements.

However, DSLs also have several drawbacks:

DSLs are difficult to create
: Although writing a DSL is “cool,” the effort shouldn't be underestimated. First, the implementation techniques can be nontrivial (see the following example). Second, good DSLs are harder to design than traditional APIs. The latter tend to follow language idioms for API design, where uniformity is important and easy to follow. In contrast, because each DSL is a unique language, we are free to create idioms that reflect the unique features of the target domain. The greater latitude makes it harder to find the best abstractions.

DSLs are hard to maintain
: DSLs can require more maintenance over the long term as the domain changes, because of the nontrivial implementation techniques used. Implementation simplicity is often sacrificed for a better user experience.

However, a well-designed DSL can be a powerful tool for building flexible and robust applications, *if* it will be used frequently.

From the implementation point of view, DSLs are classified as *internal* and *external*.

An *internal* (or *embedded*) DSL is an idiomatic way of writing code in a general-purpose programming language, like Scala. No special-purpose parser is needed. In contrast, an *external* DSL is a custom language with its own custom grammar and parser.

Internal DSLs can be easier to create because they don't require a special-purpose parser. On the other hand, the constraints of the underlying language limit the options for expressing domain concepts. External DSLs remove this constraint. You can design the language any way you want, as long as you can write a reliable parser for it. Using a custom parser can be challenging, too. Returning good error messages to the user has always been a challenge for parser writers.

Examples: XML and JSON DSLs for Scala

A decade ago, XML was the *lingua franca* of machine-to-machine communication on the Internet. JSON has been usurping that role more recently. Scala's XML support is implemented partly as a library, with some built-in syntax support. Both are now moving toward deprecation, to simplify the language and to make it easier for third-party libraries to be used instead. In Scala 2.11, the XML support was extracted into a separate module from the rest of the library (see the Scaladocs.) Our `sbt` build includes this module in its dependencies, so we can use it in this section.

Let's briefly explore working with XML in Scala to see the DSL it implements. The main types we'll see are `scala.xml.Elem` and `scala.xml.Node`:

```
// src/main/scala/progscala2/dsls/xml/reading.sc
import scala.xml._
// ❶

val xmlAsString = "<sammich>...</sammich>"                              //
❷
val xml1 = XML.loadString(xmlAsString)

val xml2 =
// ❸
<sammich>
  <bread>wheat</bread>
  <meat>salami</meat>
  <condiments>
    <condiment expired="true">mayo</condiment>
    <condiment expired="false">mustard</condiment>
  </condiments>
</sammich>

for {
// ❹
  condiment <- (xml2 \\ "condiment")
  if (condiment \ "@expired").text == "true"
           "the ${condiment.text} has
} println(sexpired!"                            )

def isExpired(condiment: Node): String =
// ❺
  condiment.attribute("expired") match {
    case Some(Nil) | None => "unknown!"
    case Some(nodes) => nodes.head.text
  }

xml2 match {
// ❻
  case <sammich>{ingredients @ _*}</sammich> => {
    for {
      condiments @ <condiments>{_*}</condiments> <- ingredients
      cond <- condiments \ "condiment"
               "  condiment: ${cond.text} is expired?
    } println(s${isExpired(cond)}"
)
  }
}
```

❶

Import the public API declarations from the `scala.xml` package.

❷

Define a string containing XML and parse it into a `scala.xml.Elem`. The `XML` object can read from a variety of sources, including URLs.

❸

Use an *XML literal* to define a scala.xml.Elem.

❹

Iterate through the XML and extract fields. The xml \ "foo" only matches on child nodes, while the xml \\ "foo" operator traverses deeper in the tree of nodes, if necessary. XPath expressions are supported, such as the @expired expression that finds attributes with that name.

❺

A helper method to find all occurrences of the attribute expired in the condiment node. If an empty sequence or None is returned, return “unknown!” Otherwise, grab the first one in the sequence and return its text, which should be “true” or “false.”

❻

Pattern match with XML literals. This expression extracts the ingredients, then a sequence of the condiment tags, then it extracts each condiment and prints data about it.

The XML object supports a few ways to save XML to a file or to a java.io.Writer:

```
// src/main/scala/progscala2/dsls/xml/writing.sc

XML.save("sammich.xml", xml2, "UTF-8")
// ❶
```

❶

Write the XML starting at the xml2 node to the file sammich.xml in this directory, using UTF-8 encoding.

Scala added limited support for JSON parsing in its *parser combinator* library, which we’ll explore in External DSLs with Parser Combinators. There are now many excellent JSON libraries for Scala, as well as for Java, so the built-in JSON support should only be considered for limited needs. So, which alternative should you use? If you’re using a major framework already, consult the documentation for its preferred choice. Otherwise, because the landscape is changing rapidly, your best bet is to search for options that seem right for your needs.

Internal DSLs

Several features of Scala syntax support creation of *internal* (embedded) DSLs:

Flexible rules for names

Because you can use almost any characters in a name, it’s easy to create names that fit the domain, like algebraic symbols for types with corresponding properties. For example, if you have a Matrix type, you can implement matrix multiplication with a * method.

Infix and postfix notation

Defining a * method wouldn’t make much sense if you couldn’t use infix notation, e.g., matrix1 * matrix2. Postfix notation, e.g., 1 minute, is also useful.

Implicit arguments and default argument values

Both features reduce boilerplate and hide complex details, such as a context that has to be passed to every method in the DSL, but can be handled instead with an implicit value. Recall that many Future methods take an implicit ExecutionContext.

Type classes

  A related use of implicits are conversions to “add” methods to existing types. For example, the `scala.concurrent.duration` package has implicit conversions for numbers that allow you to write `1.25 minutes`, which returns a `FiniteDuration` instance equal to 75 seconds.

Dynamic method invocation

  As we discussed in Chapter 19, the `Dynamic` trait makes it possible for an object to accept almost any apparent method or field invocation, even when the type has no method or field defined with that name.

Higher-order functions and by-name parameters

  Both enable custom DSLs to look like native control constructs, like the `continue` example we saw in Call by Name, Call by Value.

Self-type annotations

  Nested parts of a DSL implementation can refer to an instance in an enclosing scope if the latter has a self-type annotation visible to the nested parts. This could be used to update a state object in the enclosing scope, for example.

Macros

  Some advanced scenarios can be implemented using the new *macros* facility, which we’ll learn about in Chapter 24.

Let’s create an internal DSL for a payroll application that computes an employee’s paycheck every pay period (two weeks). The DSL will compute the *net* salary, by subtracting the *deductions* from the *gross* salary, such as taxes, insurance premiums, retirement fund contributions, etc.

Let’s begin with some common types we’ll use in both the internal and external DSLs:

```
// src/main/scala/progscala2/dsls/payroll/common.scala
package progscala2.dsls.payroll

object common {
  sealed trait Amount { def amount: Double }
// ❶

  case class Percentage(amount: Double) extends Amount {
    override def toString = s"$amount%"
  }

  case class Dollars(amount: Double) extends Amount {
    override def toString = s"$$$amount"
  }

  implicit class Units(amount: Double) {
// ❷
    def percent = Percentage(amount)
    def dollars = Dollars(amount)
  }

  case class Deduction(name: String, amount: Amount) {                 //
❸
                                 "$name:
    override def toString = s$amount"
  }

  case class Deductions(
// ❹
    name: String,
    divisorFromAnnualPay: Double = 1.0,
    var deductions: Vector[Deduction] = Vector.empty) {

    def gross(annualSalary: Double): Double =
// ❺
      annualSalary / divisorFromAnnualPay

    def net(annualSalary: Double): Double = {
      val g = gross(annualSalary)
      (deductions foldLeft g) {
        case (total, Deduction(deduction, amount)) => amount match {
          case Percentage(value) => total - (g * value / 100.0)
          case Dollars(value) => total - value
        }
      }
    }

    override def toString =
// ❻
       "$name                                    "\n     "\n
      sDeductions:"        + deductions.mkString("     , "     , "")
  }
}
```

❶

A small sealed type hierarchy that encapsulates a deduction “amount” that is either a percentage deduction from the gross or a fixed dollar amount.

❷

An implicit class that handles conversion from `Double` to the correct `Amount` subtype. It is only used in the internal DSL.

❸

A type for a single deduction with a `name` and an `amount`.

❹

A type for all the deductions. It also holds a name (e.g., “Biweekly”) and a “divisor” used to calculate the period’s gross pay from the annual gross pay.

❺

Once the deductions are constructed, return the gross and net for the pay period.

❻

Most of the `toString` methods are overridden to return the format we want.

Here is the start of the internal DSL, including a `main` that demonstrates the DSL syntax:

```
//
src/main/scala/progscala2/dsls/payroll/internal/dsl.scala
package progscala2.dsls.payroll.internal
import scala.language.postfixOps
// ❶
import progscala2.dsls.payroll.common._

object Payroll {
// ❷

  import dsl._
// ❸

  def main(args: Array[String]) = {
    val biweeklyDeductions = biweekly { deduct =>
// ❹
      deduct federal_tax          (25.0  percent)
      deduct state_tax            (5.0   percent)
      deduct insurance_premiums   (500.0 dollars)
      deduct retirement_savings   (10.0  percent)
    }

    println(biweeklyDeductions)
// ❺
    val annualGross = 100000.0
    val gross = biweeklyDeductions.gross(annualGross)
    val net   = biweeklyDeductions.net(annualGross)
           "Biweekly pay (annual: $${annualGross}%.2f):
    print(f"                                             )
            "Gross: $${gross}%.2f, Net:
    println(f$${net}%.2f"                             )
  }
}
```

❶

We want to use postfix expressions, e.g., `20.0 dollars`.

❷

Object to test the DSL.

❸

Import the DSL, which we'll see in a moment.

❹

The DSL in action. Hopefully a business stakeholder can easily understand the rules expressed here and perhaps even edit them. To be clear, *this is Scala syntax*.

❺

Print the deductions, then compute the net pay for the biweekly payroll.

The output of this program is as follows (the test `progscala2.dsls.payroll.internal.DSLSpec` uses *ScalaCheck* for more exhaustive verification):

```
Biweekly Deductions:
  federal taxes: 25.0%
  state taxes: 5.0%
  insurance premiums: $500.0
  retirement savings: 10.0%
Biweekly pay (annual: $100000.00): Gross: $3846.15, Net:
$1807.69
```

Now let’s see how it’s implemented:

```
object dsl {
//  ❶

  def biweekly(f: DeductionsBuilder => Deductions) =                 //
❷
    f(new DeductionsBuilder("Biweekly", 26.0))

  class DeductionsBuilder(
//  ❸
    name: String,
    divisor: Double = 1.0,
    deducts: Vector[Deduction] = Vector.empty) extends Deductions(
      name, divisor, deducts) {

    def federal_tax(amount: Amount): DeductionsBuilder = {           //
❹
                                                 "federal
      deductions = deductions :+ Deduction(taxes"         , amount)
      this
    }

    def state_tax(amount: Amount): DeductionsBuilder = {
                                                 "state
      deductions = deductions :+ Deduction(taxes"       , amount)
      this
    }

    def insurance_premiums(amount: Amount): DeductionsBuilder = {
      deductions = deductions :+ Deduction("insurance premiums", amount)
      this
    }

    def retirement_savings(amount: Amount): DeductionsBuilder = {
                                                 "retirement
      deductions = deductions :+ Deduction(savings"           , amount)
      this
    }
  }
}
```

❶

Wrap the public DSL pieces in an object.

❷

The method `biweekly` is the entry point for defining deductions. It constructs an empty `DeductionsBuilder` object that will be mutated in place (the easiest design choice) to add new `Deduction` instances.

❸

Build the `Deductions`, which it subclasses for convenience. The end user only sees the `Deductions` object, but the builder has extra methods for sequencing expressions.

❹

The first of the four kinds of deductions supported. Note how it updates the `Deductions` instance in place.

The DSL works as written, but I would argue that it's far from perfect. Here are some issues:

It relies heavily on Scala syntax tricks
: It exploits infix notation, function literals, etc. to provide the DSL, but it would be easy for a user to break the code by adding periods, parentheses, and other changes that seem harmless.

The syntax uses arbitrary conventions
: Why are the curly braces and parentheses where they are? Why is the `deduct` argument needed in the anonymous function?

Poor error messages
: If the user enters invalid syntax, Scala error messages are presented, not domain-centric error messages.

The DSL doesn't prevent the user from doing the wrong thing
: Ideally, the DSL would not let the user invoke any construct in the wrong context. Here, too many constructs are visible in the `dsl` object. Nothing prevents the user from calling things out of order, constructing instances of implementation constructs (like `Percentage`), etc.

It uses mutable instances
: Maybe this isn't so bad, unless you're a purist. A DSL like this is not designed to be high performance nor would you run it in a multithreading context. The mutability simplifies the implementation without serious compromises.

Most of these issues could be be fixed with more effort.

Some of my favorite examples of internal DSLs are the popular Scala testing libraries, *ScalaTest*, *Specs2*, and *ScalaCheck*. They provide great examples of DSLs that work well for developers, making the effort of creating these DSLs justified.

External DSLs with Parser Combinators

When you write a parser for an external DSL, you can use a parser generator tool like Antlr. However, the Scala library includes a *parser combinator* library that can be used for parsing most external DSLs that have a context-free grammar. An attractive feature of this library is the way it defines an internal DSL that makes parser definitions look very similar to familiar grammar notations, like Extended Backus-Naur Form (EBNF).

Scala 2.11 moved the parser combinators to a separate JAR file, so it's now optional. There are other libraries that often provide better performance, such as Parboiled 2. We'll use Scala's library for our example. Other libraries offer similar DSLs.

We have included the parser combinators library in the `sbt` build dependencies (see the Scaladocs).

About Parser Combinators

Just as the collection combinators we already know construct data transformations, parser combinators are building blocks for parsers. Parsers that handle specific bits of input, such as floating-point numbers, integers, etc., are combined together to form parsers for larger expressions. A good parser library supports sequential and alternative cases, repetition, optional terms, etc.

A Payroll External DSL

We'll reuse the previous example, but with a simpler grammar, because our external DSL does not have to be valid Scala syntax. Other changes will make parser construction easier, such as adding commas between each deduction declaration.

As before, let's start with the imports and `main` routine:

```
// src/main/scala/progscala2/dsls/payroll/parsercomb/dsl.scala
package progscala2.dsls.payroll.parsercomb
import scala.util.parsing.combinator._
import progscala2.dsls.payroll.common._
// ❶

object Payroll {

  import dsl.PayrollParser
// ❷

  def main(args: Array[String]) = {
    val input = """biweekly {
// ❸
      federal tax          20.0  percent,
      state tax            3.0   percent,
      insurance premiums   250.0 dollars,
      retirement savings   15.0  percent
    }"""
    val parser = new PayrollParser
// ❹
    val biweeklyDeductions = parser.parseAll(parser.biweekly, input).get

    println(biweeklyDeductions)
// ❺
    val annualGross = 100000.0
    val gross = biweeklyDeductions.gross(annualGross)
    val net   = biweeklyDeductions.net(annualGross)
           "Biweekly pay (annual: $${annualGross}%.2f): 
    print(f"                                                    )
             "Gross: $${gross}%.2f, Net: 
    println(f$${net}%.2f"                                    )
  }
}
```

❶

Use some of the `common` types again.

❷

Use the "root" parser for deductions.

❸

The input. Note that this time the input is a `String`, not an idiomatic Scala expression.

❹

Create a parser instance and use it by calling `biweekly`, which returns a parser for the entire DSL. The `parseAll` method returns a `Parsers.ParseResult`. To get the `Deductions`, we have to call `get`.

❺

Print the output just like the previous example. The deduction numbers are different from before. Hence the *net* will be different.

Here is the parser definition:

```
object dsl {

  class PayrollParser extends JavaTokenParsers {                      //
❶

    /** @return Parser[(Deductions)]
    */
    def biweekly = "biweekly" ~> "{" ~> deductions <~ "}" ^^ { ds => // ❷
      Deductions("Biweekly", 26.0, ds)
    }

    /** @return Parser[Vector[Deduction]]
    */
    def deductions = repsep(deduction, ",") ^^ { ds =>                //
❸

      ds.foldLeft(Vector.empty[Deduction]) (_ :+ _)
    }

    /** @return Parser[Deduction]
    */
    def deduction = federal_tax | state_tax | insurance | retirement //
❹

    /** @return Parser[Deduction]
    */
    def federal_tax = parseDeduction("federal", "tax")                //
❺

    def state_tax   = parseDeduction("state", "tax")
    def insurance   = parseDeduction("insurance", "premiums")
    def retirement  = parseDeduction("retirement", "savings")

    private def parseDeduction(word1: String, word2: String) =        //
❻

      word1 ~> word2 ~> amount ^^ {
        amount => Deduction(s"${word1} ${word2}", amount)
      }

    /** @return Parser[Amount]
    */
    def amount = dollars | percentage
//  ❼

    /** @return Parser[Dollars]
    */
    def dollars = doubleNumber <~ "dollars" ^^ { d => Dollars(d) }

    /** @return Parser[Percentage]
    */
    def percentage = doubleNumber <~ "percent" ^^ { d => Percentage(d) }

    def doubleNumber = floatingPointNumber ^^ (_.toDouble)
  }
}
```

❶

The class defining the grammar and parser, through methods.

❷

The top-level parser, created by building up smaller parsers. The entry method `biweekly` returns a `Parser[Deductions]`, which is a parser capability of parsing a string for a complete deductions specification. It returns a `Deductions` object. We'll discuss the syntax in a moment.

❸

Parse a comma-separated list of deductions. Adding the requirement to use a comma simplifies the parser implementation. The `repsep` method parses an arbitrary number of deduction expressions.

❹

Recognize four possible deductions.

❺

Call a helper function to construct the four deduction parsers.

❻

The helper method for the four deductions.

❼

Parse the amount, a double literal followed by `dollars` or `percent`. A corresponding `Amount` instance is constructed.

Let's look at `biweekly` more closely. Here it is rewritten a bit to aid the discussion:

```
"biweekly" ~> "{" ~> deductions <~ "}"
//  ❶
  ^^ { ds => Deductions("Biweekly", 26.0, ds) }
//  ❷
```

❶

Find three *terminal tokens*, `biweekly`, `{`, and `}`, with the results of evaluating the `deductions` *production* between the `{…}`. The arrow-like operators (actually methods, as always), `~>` and `<~`, mean drop the token on the side of the `~`. So the literals are dropped and only the result of `deductions` is retained.

❷

The `^^` separates the left side (*reduction tokens*) from the right side (*grammar rule*) for the production. The *grammar rule* takes as arguments the tokens retained. If there is more than one, a partial function literal is used of the form `{ case t1 ~ t2 ~ t2 => ... }`, for example. In our case, `ds` is a `Vector` of `Deduction` instances, which is used to construct a `Deductions` instance.

Note that `DeductionsBuilder` in the internal DSL is not needed here. See the test `progscala2.dsls.payroll.parsercomb.DSLSpec`, which uses *ScalaCheck*, for exhaustive verification.

Internal Versus External DSLs: Final Thoughts

Let's compare the internal and external DSLs the user writes. Here is the internal DSL again:

```
val biweeklyDeductions = biweekly { deduct
=>
  deduct federal_tax          (25.0
percent)
  deduct state_tax            (5.0
percent)
  deduct insurance_premiums   (500.0
dollars)
  deduct retirement_savings   (10.0
percent)
}
```

Here is the external DSL again:

```
val input = """biweekly {
  federal tax         20.0
percent,
  state tax           3.0
percent,
  insurance premiums  250.0
dollars,
  retirement savings  15.0  percent
}"""
```

The external DSL is simpler, but the user must embed the DSL in strings. Hence, code completion, refactoring, color coding, and other IDE features aren't available.

On the other hand, the external DSL is easier (and actually more fun) to implement. It should have less fragility from reliance on Scala parsing tricks.

You'll have to weigh which trade-offs make the most sense for your situation. If the DSL is "close-enough" that it can be implemented internally with reasonable effort and robustness, the user experience will generally be better. It's clearly the best choice for the test libraries mentioned earlier. If the DSL is too far removed from Scala syntax, perhaps because it's a well-known language, like SQL, using an external DSL with quoted strings is probably best.

Recall that we can implement our own string interpolators (see Build Your Own String Interpolator). This is a useful way to encapsulate a parser built with combinators behind a slightly easier syntax. For example, if you implement a SQL parser of some sort, let the user invoke it with `sql"SELECT * FROM table WHERE …;"`, rather than having to use the parser API calls explicitly like we did.

Recap and What's Next

It's tempting to create DSLs with abandon. DSLs in Scala can be quite fun to work with, but don't underestimate the effort required to create robust DSLs that meet your clients' usability needs, while at the same time requiring reasonable effort for long-term maintenance and support.

In the next chapter, we'll explore the Scala tools and and libraries.