The busy Java developer's guide to Scala: Building a calculator, Part 3

Combining Scala's parser combinators and case classes

Skill Level: Introductory

Ted Neward (ted@tedneward.com)
Principal
Neward & Associates

25 Nov 2008

Domain-specific languages (DSLs) have become a hot topic; much of the buzz around functional languages is their ability to build such languages. After having established both an AST scheme and a basic front-end parser designed to take text and produce a graph of objects suitable for interpretation, in this article the author thinks it's time to wire everything together into a seamless -- if somewhat trivial at this point -- whole. Then he'll turn around and suggest some extensions that could be made to the language and interpreter.

Welcome back, dauntless readers. The journey into Scala's language and library support continues as this month we turn around and take the calculator DSL and finally "finish it off." The DSL itself is somewhat trivial — a simple calculator that so far only supports the four cardinal mathematic operations — but remember, the goal is to create something that will be extensible and flexible and easily enhanced to support new functionality later.

When we last left our heroes...

To recap, our DSL at the moment is a bit disconnected. We have an Abstract Syntax Tree made up of a number of case classes ...

Listing 1. The back end (AST)

```
package com.tedneward.calcdsl
{
  // ...

  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Variable(name : String) extends
Expr
  private[calcdsl]  case class Number(value : Double) extends
Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg :
Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String, left
: Expr, right : Expr)
   extends Expr

}
```

... which we can walk to provide interpreter-like behavior with some optimizations around simplification of mathematical expressions ...

Listing 2. The back end (interpreter)

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    def simplify(e: Expr): Expr = {
      // first simplify the subexpressions
      val simpSubs = e match {
        // Ask each side to simplify
        case BinaryOp(op, left, right) => BinaryOp(op, simplify(left), simplify(right))
        // Ask the operand to simplify
        case UnaryOp(op, operand) => UnaryOp(op, simplify(operand))
        // Anything else doesn't have complexity (no operands to simplify)
        case _ => e
      }

      // now simplify at the top, assuming the components are already simplified
      def simplifyTop(x: Expr) = x match {
        // Double negation returns the original value
        case UnaryOp("-", UnaryOp("-", x)) => x

        // Positive returns the original value
        case UnaryOp("+", x) => x

        // Multiplying x by 1 returns the original value
        case BinaryOp("*", x, Number(1)) => x

        // Multiplying 1 by x returns the original value
        case BinaryOp("*", Number(1), x) => x

        // Multiplying x by 0 returns zero
        case BinaryOp("*", x, Number(0)) => Number(0)

        // Multiplying 0 by x returns zero
        case BinaryOp("*", Number(0), x) => Number(0)

        // Dividing x by 1 returns the original value
        case BinaryOp("/", x, Number(1)) => x

        // Dividing x by x returns 1
```

```
        case BinaryOp("/", x1, x2) if x1 == x2 => Number(1)

        // Adding x to 0 returns the original value
        case BinaryOp("+", x, Number(0)) => x

        // Adding 0 to x returns the original value
        case BinaryOp("+", Number(0), x) => x

        // Anything else cannot (yet) be simplified
        case e => e
      }
      simplifyTop(simpSubs)
    }

    def evaluate(e : Expr) : Double =
    {
      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

... and we have a text parser, built using Scala's parser combinator library to parse simple math expressions ...

Listing 3. The front end

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    object ArithParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    // ...
  }
}
```

... but which, when it parses, produces a collection of Strings and Lists owing to the fact that the parser combinators are currently written to return `Parser[Any]` types, essentially leaving it up to the parser to return whatever it feels like (which, as we can see, is a collection of Strings and Lists).

For the DSL to be successful in any meaningful way, the parser needs to instead return objects from our AST so that when parsing is complete, the execution engine can grab the tree and `evaluate()` it. Toward that end, we need to change the parser combinator implementation to produce different objects during parse.

Cleaning up the grammar

The first change I've decided to make to the parser is one of actual grammar. In the original parser, expressions like "5 + 5 + 5" were acceptable owing to the `rep()` combinator in the grammar definition for expressions (`expr`) and terms (`term`). But after some thought about extensions, I realized this potentially raises some issues with associativity and operator precedence with future operations that could be eliminated by requiring pairs of operations to be given explicit precedence using parentheses. So the first change will be to change the grammar to require "()" around all expressions.

In retrospect, this was something I should have done from the first; it's a fact that it's generally easier to relax restrictions than add them later (in the event that those restrictions turn out to be unnecessary or gratuitous) but a lot harder to work out some of the thorny issues around operator precedence and associativity. If you're not sure what operator precedence and associativity is, go look for a general discussion of it; for a general overview of how complex the situation can get, consider the Java language itself and the various operators it supports (as showcased in the Java Language Specification) or some of the associativity puzzles (from *Java Puzzlers* by Bloch and Gafter). It's not a pretty picture.

So, just to take things in incremental steps, the first thing is to test the grammar again:

Listing 4. Taking parens

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    object OldAnyParser extends JavaTokenParsers
    {
      def expr: Parser[Any] = term ~ rep("+"~term | "-"~term)
      def term : Parser[Any] = factor ~ rep("*"~factor | "/"~factor)
      def factor : Parser[Any] = floatingPointNumber | "("~expr~")"

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }
    object AnyParser extends JavaTokenParsers
```

```
    {
      def expr: Parser[Any] = (term~"+"~term) | (term~"-"~term) | term
      def term : Parser[Any] = (factor~"*"~factor) | (factor~"/"~factor) | factor
      def factor : Parser[Any] = "(" ~> expr <~ ")" | floatingPointNumber

      def parse(text : String) =
      {
        parseAll(expr, text)
      }
    }

    // ...
  }
}
```

I've renamed the older parser to be `OldAnyParser` and left it in for comparison's sake; the new grammar is given by `AnyParser`; notice how it defines `exprs` to be either a `term + term`, a `term - term`, or a stand-alone `term`, and so on. The other major change is in the definition of `factor` which now uses a different combinator, ~> and <~, to effectively throw away the `(` and `)` characters when it encounters them.

Because this is just a temporary step, I'm not going to bother creating a series of unit tests to text all the possibilities. Still, I want to make sure the grammar parse results are going to be what I expect so I'll cheat here for a moment and write a test that's not really a test:

Listing 5. Cheater, cheater, but still testing the parser eater

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    @Test def parse =
    {
      import Calc._

      val expressions = List(
        "5",
        "(5)",
        "5 + 5",
        "(5 + 5)",
        "5 + 5 + 5",
        "(5 + 5) + 5",
        "(5 + 5) + (5 + 5)",
        "(5 * 5) / (5 * 5)",
        "5 - 5",
        "5 - 5 - 5",
        "(5 - 5) - 5",
        "5 * 5 * 5",
        "5 / 5 / 5",
        "(5 / 5) / 5"
      )

      for (x <- expressions)
        System.out.println(x + " = " + AnyParser.parse(x))
```

```
    }
  }
}
```

Remember, folks, this is purely for pedagogical purposes (fancy author-speak for "You really don't want to do this for production code, but because I'm not writing production code, I get to cheat. You don't. Really.") Running the tests, however, will pipe a number of results to the standard-out section of the unit test results file and show that the non-parenthesized expressions (`5 + 5 + 5`) will fail while the parenthesized ones succeed. Marvelous.

Don't forget to comment out the parse test. Or better yet, delete it entirely. It's a hack and as we all know, a true Jedi only uses the Source for knowledge and defense, never for a hack.

Cleaning up the grammar

Now we need to change the definition of the various combinators again (yes, again). Recall from the last article that each of the functions `expr`, `term`, and `factor` are essentially BNF statements, but notice that the return of each is a genericized type of Parser parameterizing on `Any` (an ultimate supertype in the Scala type system that essentially serves the purpose its name implies: to indicate a potential type or reference that can contain anything); this indicates that the combinator is free to return as it wishes. As we've already seen, by default, the parser will return either a String or a List. (We can see that from the results in that run of the tests with the hack test in it too, if you're not convinced.)

To change it to produce instances of our case class AST hierarchy (`Expr` objects), the return type of the combinators has to be changed to be `Parser[Expr]`. If you make that change on its own, it will fail to compile; the three combinators know how to take in Strings but not how to produce `Expr` objects from what's parsed. To do that, we make use of one more combinator, the `^^` combinator, which takes an anonymous function as a parameter, passing the results of the parse into the anonymous function as a parameter.

If you're like many Java developers, that'll take a second to parse (pardon the pun), so let's look at it in action:

Listing 6. The production combinator

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    object ExprParser extends JavaTokenParsers
```

```
    {
      def expr: Parser[Expr] =
        (term ~ "+" ~ term) ^^ { case lhs~plus~rhs => BinaryOp("+", lhs, rhs) } |
        (term ~ "-" ~ term) ^^ { case lhs~minus~rhs => BinaryOp("-", lhs, rhs) } |
        term

      def term: Parser[Expr] =
        (factor ~ "*" ~ factor) ^^ { case lhs~times~rhs => BinaryOp("*", lhs, rhs) } |
        (factor ~ "/" ~ factor) ^^ { case lhs~div~rhs => BinaryOp("/", lhs, rhs) } |
        factor

      def factor : Parser[Expr] =
        "(" ~> expr <~ ")" |
        floatingPointNumber ^^ {x => Number(x.toFloat) }

      def parse(text : String) = parseAll(expr, text)
    }

    def parse(text : String) =
      ExprParser.parse(text).get

    // ...
  }

  // ...
}
```

The ^^ combinator takes an anonymous function to which the results of the parse (for example, assuming the `5 + 5` input, the results will be `((5~+)~5)`) will be passed to be cracked apart and yield a resulting object — in this case, a `BinaryObject` of the appropriate type. Notice again the power of pattern matching; I'm binding the left-hand part of the expression to the `lhs` instance, the + portion to the (unused) `plus` instance, and the right-hand side of the expression to `rhs`, then I promptly use both `lhs` and `rhs` to occupy the left- and right-hand sides of the `BinaryOp` constructor, respectively.

Running the code now (you did remember to comment out the hack test, right?) against the unit test suite yields all positive results: the various expressions we tried before no longer fail because now the parser produces `Expr`-derived objects. Having said that, it would be irresponsible to not exercise the parser further, so let's add some more tests (including the tests that I sort of cheated from the parser earlier):

Listing 7. Testing the parser (for real this time)

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    @Test def parseAnExpr1 =
      assertEquals(
        Number(5),
        Calc.parse("5")
      )
```

```
    @Test def parseAnExpr2 =
      assertEquals(
        Number(5),
        Calc.parse("(5)")
      )
    @Test def parseAnExpr3 =
      assertEquals(
        BinaryOp("+", Number(5), Number(5)),
        Calc.parse("5 + 5")
      )
    @Test def parseAnExpr4 =
      assertEquals(
        BinaryOp("+", Number(5), Number(5)),
        Calc.parse("(5 + 5)")
      )
    @Test def parseAnExpr5 =
      assertEquals(
        BinaryOp("+", BinaryOp("+", Number(5), Number(5)), Number(5)),
        Calc.parse("(5 + 5) + 5")
      )
    @Test def parseAnExpr6 =
      assertEquals(
        BinaryOp("+", BinaryOp("+", Number(5), Number(5)), BinaryOp("+", Number(5),
                 Number(5))),
        Calc.parse("(5 + 5) + (5 + 5)")
      )

    // other tests elided for brevity
  }
}
```

Readers are encouraged to add a few just to make sure I didn't leave out an edge case. (There's nothing like pair programming when you're pairing with the rest of the Internet, let me tell you.)

Tying up the loose ends

Given that the parser now works the way we want it — that is producing the AST — it only remains now to tie the parser's results up against the evaluation of the AST objects and see what emerges. Doing so is almost anticlimactic; it just means adding the code in Listing 8 to `Calc` ...

Listing 8. It's finished, Igor, finished!

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    def evaluate(text : String) : Double = evaluate(parse(text))
  }
}
```

... and adding a simple test to ensure that `evaluate("1+1")` returns 2.0 ...

Listing 9. All this, just to see if 1 + 1 is 2?

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    // ...

    @Test def add1 =
      assertEquals(Calc.evaluae("1 + 1"), 2.0)
  }
}
```

... and then running it. Seriously ... run it. Feels good, doesn't it?

Extending the language

Given the fact that the same calculator DSL could have been written in plain Java code without having to go to the trouble I did (just evaluate each fragment recursively without building a whole AST and so on), it may seem like this is another case of a language or tool trying to find a problem to solve. But the real power of building a language this way comes when extending or scaling it up to include more features later.

For example, let's add a new operator to the language, the ^ operator, which will perform exponentiation; in other words, *2 ^ 2* is *2 squared* or 4. To add this to the language requires some simple steps.

First, you have to consider whether a change to the AST needs to be made. In this case, the exponentiation operator is another form of binary operator so the existing `BinaryOp` case class serves. No AST changes are necessary.

Next, the `evaluate` function has to be modified to do the right thing with a `BinaryOp("^", x, y)`; this is a simple matter of adding a nested function (because it's not necessary to be seen outside of that function anyway) to handle the actual calculation of exponents and then adding the necessary line to the pattern match, like so:

Listing 10. Uh ... hang on a second, Igor ...?

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...
```

```
    def evaluate(e : Expr) : Double =
    {
      def exponentiate(base : Double, exponent : Double) : Double =
        if (exponent == 0)
          1.0
        else
          base * exponentiate(base, exponent - 1)

      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
        case BinaryOp("^", x1, x2) => exponentiate(evaluate(x1), evaluate(x2))
      }
    }
  }
}
```

Notice how here the fact that we added exponentiation to the system has been done in effectively six lines of code with zero surface change to the rest of the `Calc` class. That's encapsulation!

(In my efforts to create the simplest exponentation function that could possibly work, I have deliberately created a version that has a pretty serious, albeit intentional, bug in it -- this was done to keep focus on the language, not the implementation. That said, kudos and credit to the first reader who finds it, writes the unit test that exposes it, and offers a fixed version.)

Before I go and add it to the parser though, let's exercise the code (also known as "keepin' it real" in agile circles) by writing a few tests to make sure that the exponentiation parts work correctly:

Listing 11. You're such a square

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...

    @Test def evaluateSimpleExp =
    {
      val expr =
        BinaryOp("^", Number(4), Number(2))
      val results = Calc.evaluate(expr)
      // (4 ^ 2) => 16
      assertEquals(16.0, results)
    }
    @Test def evaluateComplexExp =
    {
      val expr =
        BinaryOp("^",
          BinaryOp("*", Number(2), Number(2)),
          BinaryOp("/", Number(4), Number(2)))
      val results = Calc.evaluate(expr)
      // ((2 * 2) ^ (4 / 2)) => (4 ^ 2) => 16
```

```
      assertEquals(16.0, results)
    }
  }
}
```

Running this code verifies that exponentiation works (modulo the bug I mentioned earlier), so half of the battle is now complete.

The last change needed is to modify the grammar to accept the new exponentiation operator; because exponentiation is of the same level of precedence as multiplication and division, it's most easily accomplished by putting it in the `term` combinator:

Listing 12. It's finished, Igor. This time I mean it!

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    // ...

    object ExprParser extends JavaTokenParsers
    {
      def expr: Parser[Expr] =
        (term ~ "+" ~ term) ^^ { case lhs~plus~rhs => BinaryOp("+", lhs, rhs) } |
        (term ~ "-" ~ term) ^^ { case lhs~minus~rhs => BinaryOp("-", lhs, rhs) } |
        term

      def term: Parser[Expr] =
        (factor ~ "*" ~ factor) ^^ { case lhs~times~rhs => BinaryOp("*", lhs, rhs) } |
        (factor ~ "/" ~ factor) ^^ { case lhs~div~rhs => BinaryOp("/", lhs, rhs) } |
        (factor ~ "^" ~ factor) ^^ { case lhs~exp~rhs => BinaryOp("^", lhs, rhs) } |
        factor

      def factor : Parser[Expr] =
        "(" ~> expr <~ ")" |
        floatingPointNumber ^^ {x => Number(x.toFloat) }

      def parse(text : String) = parseAll(expr, text)
    }

  // ...
  }
}
```

Of course, we need a few tests to exercise the parser ...

Listing 13. Are you a square, too?

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...

    @Test def parseAnExpr17 =
```

```
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("2 ^ 2")
      )
    @Test def parseAnExpr18 =
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("(2 ^ 2)")
      )
    @Test def parseAnExpr19 =
      assertEquals(
        BinaryOp("^", Number(2),
          BinaryOp("+", Number(1), Number(1))),
        Calc.parse("2 ^ (1 + 1)")
      )
    @Test def parseAnExpr20 =
      assertEquals(
        BinaryOp("^", Number(2), Number(2)),
        Calc.parse("2 ^ (2)")
      )
  }
}
```

... which when run, pass, thus leaving us with the final test to see if everything is wired up correctly:

Listing 14. From String to square!

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    // ...

    @Test def square1 =
      assertEquals(Calc.evaluate("2 ^ 2"), 4.0)
  }
}
```

Success!

About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this developerWorks series, you'll learn what all the recent hype is about and see some of Scala's linguistic capabilities in action. Scala code and Java code will be shown side by side wherever comparison is relevant, but (as you'll discover) many things in Scala have no direct correlation to anything you've found in Java -- and therein lies much of Scala's charm! After all, if Java could do it, why bother learning Scala?

Conclusion

Clearly, this is a bit more work than is merited for a language this simple; regardless of how you feel about testing the individual parts of the language (the AST, the

parser, the simplification engine, and so on), it would have been much shorter (and possibly faster depending on how often you've done this) to simply write the language as a more straightforward Interpreter-based object, possibly even calculating the expressions on the fly rather than translating them into an AST and evaluating from there.

But consider how trivial it was to add another operator into the system and how the design of the language's implementation made it actually quite easy to extend without having to touch code in a variety of different places. In fact, we have a lot of possible enhancements that could be done to demonstrate the inherent flexibility of this approach:

- We could convert from using `Doubles` to `BigDecimals` or `BigIntegers` from the java.math package (to allow for much larger and/or more accurate calculations).
- We could add support for decimal numbers in the language (currently not supported in the parser).
- We could add new operators using words ("sin," "cos," "tan," etc.) instead of symbols.
- We could even add a variable notation ("x = y + 12") and accept a `Map` as a parameter to the `evaluate()` function containing the starting values of each of those variables.

More importantly, the DSL is entirely hidden away behind the `Calc` class as a facade, meaning that it's as trivial to call from Java code as it is from Scala. So even in projects that aren't ready to wholly adopt Scala as their first language of choice, parts of the system — those best served by a functional/object fusion language — can be written in Scala and simply made available for Java developers to use as they wish.

That's it for now; in the next set of articles, we'll go back to visiting more of Scala's language capabilities (such as generics, because parser combinators made use of them). There's still a lot of Scala left to learn, but hopefully now it's a bit more clear how you can use Scala to solve problems that had, in Java code, been much harder before now. Until next time, sports fans!

Resources

Learn

- "*The busy Java developer's guide to Scala*: Functional programming for the object oriented" (Ted Neward, developerWorks, January 2008): The first article in this series gives you an overview of Scala and explains its functional approach to concurrency among other things. Others in the series:
  - "Class action" (February 2008) details Scala's class syntax and semantics.
  - "Don't get thrown for a loop!" (March 2008) dives deep inside Scala's control structures.
  - "Of traits and behaviors" (April 2008) leverages Scala's version of Java interfaces.
  - "Implementing inheritance" (May 2008) is polymorphism done the Scala way.
  - "Collection types" (June 2008) is all "tuples, arrays, and lists," oh my!
  - "Packages and access modifiers" (July 2008) covers Scala's package and access modifier facilities and the `apply` mechanism.
  - "Building a calculator, Part 1" (August 2008) delivers the first part of the lesson from this article.
  - "Building a calculator, Part 2" (October 2008) delivers the second part of the lesson from this article, using parser combinators.
- "Functional programming in the Java language" (Abhijit Belapurkar, developerWorks, July 2004): Explains the benefits and uses of functional programming from a Java developer's perspective.
- "Scala by Example" (Martin Odersky, December 2007): A short, code-driven introduction to Scala (in PDF).
- *Programming in Scala* (Martin Odersky, Lex Spoon, and Bill Venners; Artima, December 2007): The first book-length introduction to Scala.
- *Java Puzzlers: Traps, Pitfalls, and Corner Cases* (Addison-Wesley Professional, July 2005) reveals oddities of the Java programming language through entertaining and thought-provoking programming puzzles.
- The developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download Scala: Start learning it with this series.

- SUnit: Part of the standard Scala distribution, in the *scala.testing* package.

Discuss

- Participate in the discussion forum for this content.
- developerWorks blogs: Get involved in the developerWorks community.

About the author

Ted Neward
Ted Neward is the principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.

Trademarks