The busy Java developer's guide to Scala: Building a calculator, Part 2

Scala's parser combinators

Skill Level: Introductory

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

21 Oct 2008

Domain-specific languages (DSLs) have become a hot topic; much of the buzz around functional languages is their ability to build such languages. In this latest installment of *The busy Java developer's guide to Scala*, Ted Neward tackles the problem of transforming textual input into the AST for interpretation by continuing with a simple calculator DSL that demonstrates the power of functional languages for building "external" DSLs. To parse textual input and transform it into the tree structure used by the interpreter in the last article, Ted introduces *parser combinators*, a standard Scala library designed solely for the task. (In the previous article, we built a calculator interpreter and AST.)

Recall, if you will, the predicament of our hero: In looking to create a DSL (in this case, a ridiculously simple calculator language), he had created the tree structure containing the various options available to the language:

- The binary add/subtract/multiply/divide operators
- The unary negation operator
- Numerical values

The execution engine behind it knew how to execute those operations and it even had an explicit optimization step designed to reduce the calculations necessary to yield a result.

When we last left the code, it looked like this:

Listing 1. Calculator DSL: The AST and interpreter

```
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

  object Calc
  {
    /**
     * Function to simplify (a la mathematic terms) expressions
     */
    def simplify(e : Expr) : Expr =
    {
      e match {
        // Double negation returns the original value
        case UnaryOp("-", UnaryOp("-", x)) => simplify(x)

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

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

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

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

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

    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))
      }
```

```
    }
  }
}
```

Readers of the previous article will also remember I laid out a challenge to improve the optimization step by carrying out the simplification process deeper into the tree instead of simply at the top level as the code in Listing 1 does. Lex Spoon found what I believe to be the simplest way to do the optimization: to simplify the "edges" of the tree (the operands inside each of the expressions, if any) first, then take the resulting simplification and simplify the top-level expression, shown in Listing 2:

Listing 2. Simplify, simplify ... simplified

```
/*
 * Lex's version:
 */
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)
    }
```

Thanks, Lex.

Parsing

Now comes the other half of building a DSL: We need to build a segment of code that can take some kind of textual input and turn it into an AST. This process is known more formally as *parsing* (or to be more precise, *tokenizing*, *lexing*, and parsing).

Historically, the act of creating a parser has gone down two roads:

- Building a parser by hand.
- Allowing a tool to generate the parser.

We can seek to build the parser by hand by manually pulling a character off the input stream, examining it, and taking some kind of action based not only on this character but also the other characters that came before it (and sometimes on the characters that came after it). Building parsers by hand can be quicker and easier for smaller languages, but it tends to become a difficult problem as the language gets larger.

The alternative to the hand-authored parser is to have a tool generate the parser for us; historically, this has been the province of two tools known affectionately as *lex* (because it generates a "lexer") and *yacc* ("Yet Another Compiler Compiler"). Programmers interested in writing a parser don't write the parser by hand but instead write a different source file, fed as input into "lex" which produces the very front end of the parser. This generated code is then combined with a "grammar" file, defining the basic grammatical rules of the language (in which tokens are keywords, where blocks of code can appear, and so on), and is fed into yacc to produce the parser code.

This is basic Computer Science 101 textbook, so rather than get into the details of finite state automata and details of LALR or LR parsers, I'll assume that those who are interested in the deeper details will go find the books or articles (or both!) on the subject.

In the meantime, let's explore Scala's third option for building parsers: *parser combinators*, built entirely from the functional side of Scala. Parser combinators allow us to "combine" various bits and pieces of the language together into parts that can provide a solution that doesn't require code generation and looks like a language specification to boot.

Parser combinators

To understand the point behind parser combinators, it helps to have a passing knowledge of *Becker-Naur Form (BNF)*, a way of specifying how a language will look. For example, our calculator language can be described in a BNF grammar that looks something like Listing 3:

Listing 3. Describing language

```
input ::= ws expr ws eoi;

expr ::= ws powterm [{ws '^' ws powterm}];
powterm ::= ws factor [{ws ('*'|'/') ws factor}];
factor ::= ws term [{ws ('+'|'-') ws term}];
term ::= '(' ws expr ws ')' | '-' ws expr | number;

number ::= {dgt} ['.' {dgt}] [('e'|'E') ['-'] {dgt}];
dgt ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
ws ::= [{' '|'\t'|'\n'|'\r'}];
```

where each of the elements on the left side of the statement is the name of a collection of possible inputs and the elements on the right-hand, also known as *terms*, are a series of either expressions or literal characters in optional or required combinations. (Again, the full details of BNF grammars are better described in books like Aho/Lam/Sethi/Ullman in Resources.)

The power of expressing the language in a BNF form is that from BNF, it's a short step to the Scala parser combinator; a simplified form of the BNF in Listing 3 looks something like Listing 4:

Listing 4. Simplify, simplify (again)

```
expr   ::= term {'+' term | '-' term}
term   ::= factor {'*' factor | '/' factor}
factor ::= floatingPointNumber | '(' expr ')'
```

where curly braces ({}) denote possible repetition (zero or more times) of the contents and the vertical bar (or "pipe" |) denotes an either/or relationship. Thus, reading Listing 4, a `factor` can be either a `floatingPointNumber` (whose definition isn't given here for a reason) or a left-parenthesis plus an `expr` plus a right-parenthesis.

From here, turning this into a Scala parser is insultingly straightforward, as shown in Listing 5:

Listing 5. From BNF to parsec

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

    import scala.util.parsing.combinator._

    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)
      }
    }

    def parse(text : String) =
    {
      val results = ArithParser.parse(text)
      System.out.println("parsed " + text + " as " + results + " which is a type "
       + results.getClass())
    }

        // ...
  }
}
```

The BNF is essentially replaced by a few parser combinator syntactical elements instead: spaces are replaced by the ~ method (indicating a sequence), repetitions are replaced by the `rep` method, and choices are still represented via the `|` method. The literal strings are standard literal strings.

Part of the power of this approach is seen in two parts. First, the fact that the parser extends the Scala-provided `JavaTokenParsers` base class (which itself in turn inherits from other base classes if we want a language that's not so closely aligned with the Java language's syntactic notions about the world) and second, the use of the `floatingPointNumber` preconceived combinator to handle the details of parsing a floating-point number.

While this particular grammar (that of an infix calculator) is not a difficult one to work with by any means (which is why we see it in so many demos and articles), it also would not be a difficult task to build a parser for it by hand because the close relationship between the BNF grammar and the code building the parser makes it that much easier and quicker to get a parser in place.

A parser combinator conceptual primer

To understand how this all works, we have to take a brief dive into the parser combinator implementation. In essence, each "parser" is a function or a case class that takes some kind of input and produces a "parser" as a result; for example, at the

lowest levels, the parser combinator rests on top of simple parsers that take some kind of input-reading element (a `Reader`) as input and produces something that can offer higher-level semantics (a `Parser`) as a result:

Listing 6. A basic parser

```
type Elem

type Input = Reader[Elem]

type Parser[T] = Input => ParseResult[T]

sealed abstract class ParseResult[+T]
case class Success[T](result: T, in: Input) extends ParseResult[T]
case class Failure(msg: String, in: Input) extends ParseResult[Nothing]
```

In other words, `Elem` is an abstract type representing ... well, anything that can be parsed, most commonly a text string or stream. An `Input`, then, is a `scala.util.parsing.input.Reader` wrapped around that kind of type (the square brackets indicate that `Reader` is a generic type; think of them as angle brackets if you prefer the Java-style or C++-style syntax). Then a `Parser` of type `T` is a type that takes an `Input` and produces a `ParseResult` that can be (fundamentally) of one of two types: a `Success` or a `Failure`.

There is obviously much more to the parser combinator library than just what's shown here — it's more than a few steps to get to the ~ and `rep` functions — but this gives you the basic idea of how parser combinators work. Parsers "combine" to provide higher and higher levels of abstraction over the concept of parsing (hence the name "parser combinators"; elements which combine together to provide parsing behavior).

Are we there yet, Daddy?

We're still not quite done yet. A quick test to call the parser reveals that the return of the parser isn't quite what we need for the rest of the calculator system:

Listing 7. First test fails?!?

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

        // ...

    @Test def parseNumber =
    {
      assertEquals(Number(5), Calc.parse("5"))
      assertEquals(Number(5), Calc.parse("5.0"))
    }
```

```
  }
}
```

When run, this test will fail because it turns out that the parser's `parseAll` method doesn't return our case-classed `Number` (which is somewhat reasonable since nowhere in the parser have we established the relationship of our case classes against the parser's production rules); nor does it return a collection of text tokens or ints.

Instead, the parser returns a `Parsers.ParseResult` that, remember, is either a `Parsers.Success` instance (inside of which are the results we're looking for) or else a `Parsers.NoSuccess`, `Parsers.Failure`, or `Parsers.Error` (which are all variations on the same theme: The parsing didn't work for some reason).

Assuming a successful parse, to get the actual results, we have to extract the results via the `get` method on the `ParseResult`. This means the `Calc.parse` method has to be adjusted just slightly to get this test to pass, as shown in Listing 8:

Listing 8. From BNF to parsec

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

    import scala.util.parsing.combinator._

    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)
      }
    }

    def parse(text : String) =
    {
      val results = ArithParser.parse(text)
      System.out.println("parsed " + text + " as " + results + " which is a type "
         + results.getClass())
          results.get
    }

        // ...
  }
}
```

Success! Right?

Sorry, no. Running the tests reveals that the results of the parser aren't yet in the AST types I created earlier (`expr` and its kin), but instead in a form consisting of

Lists and Strings and such. While I suppose we could turn around and parse these results into `expr` instances and interpret from there, surely another way must be available.

There is, and to understand how it works, you'll need to take a short dive into how parser combinators produce non-"standard" elements (that is, not Strings and Lists). Or to use the appropriate terminology, how a parser can produce a custom element ... in this case our AST objects. And that is a subject for the next time.

In the next installment, I'll take you on a dive into the basics of parser combinator implementation and show you how to parse the text bits into an AST for evaluation (and then later, for compilation).

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

OK, we're clearly not finished (there's parsing work yet to be done), but the basic parser semantics are in place, requiring only that we extend the parser production elements to produce AST elements.

For those readers who want to jump ahead some, check out the ^^ method described in the ScalaDocs or the section on parser combinators in *Programming in Scala*; as a warning, understand that this language is a tad trickier than the examples given in these resources may appear to be.

Of course, you could just deal with things as Strings and Lists and ignore the AST part of the world, picking apart the returned Strings and Lists and re-parsing them into AST elements. But why would you want to do that when the parser combinator library has so much more in store for you?

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.
- "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.
- 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

- 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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.