Updated: 2021-03-08

4.9
Parsing and DSLs

Northeastern University

What is a Domain-specific language?

- Any time you create a structured format for writing/reading particular objects, it’s a DSL.
- XML, JSON, etc. are *not* themselves DSLs but they can be used to create DSLs. However, you don’t need either of these types of markup language to create a DSL
- Examples:
    - Product inventories;
    - Lens defnitions;
    - Workflows…

What exactly is parsing and why do we need it?

- Any time we have some input in some sort of serialized form (such as a *CharSequence*), and we want to turn it into an object that we can reference in an expression, we need a parser. In other words, we wish to create a domain-specific language (DSL).
  - Some examples we've already encountered:
    - Creating *Rational* objects;
    - Reading the CSV file into our *Movie* program*;*
    - Reading JSON strings returned Google Finance (*lab-actors*) (or from the *Poets*);
    - Defining the rules for dealing with options (*lab-actors*).
  - In all these cases, we used a regular expression with a match:

```
object NumberPredicate {
  def apply…
  def apply(predicate: String): NumberPredicate = {
    val rPredicate = """^\s*(\w+)\s*([=<>]{1,2})\s*(-?[0-9]+\.?[0-9]*)\s*$""".r
    predicate match {
      case rPredicate(v, o, n) => apply(v, o, n)
      case _ => throw new Exception(s"predicate: $predicate is malformed")
    }
  }
}
```

Example: Rational (1)

```
implicit class RationalHelper(val sc: StringContext) extends AnyVal {
    def r(args: Any*): Rational = {
        val strings = sc.parts.iterator
        val expressions = args.iterator
        val sb = new StringBuffer()
        while(strings.hasNext) {
            val s = strings.next
            if (s.isEmpty) {
              if(expressions.hasNext)
                  sb.append(expressions.next)
          else
            throw new RationalException("r: logic error: missing expression")
            }
            else
              sb.append(s)
        }
        if(expressions.hasNext)
          throw new RationalException(s"r: ignored: ${expressions.next}")
        else
            Rational(sb.toString)
    }
}
```

Example: Rational (2)

- ```
  def apply(x: String): Rational = {
    val rRat = """^\s*(\d+)\s*(/\s*(\d+)\s*)?$""".r
    val rDec = """^-?(\d|(\d+,?\d+))*(\.\d+)?(e\d+)?$""".r
    x match {
      case rRat(n, _, null) => Rational(n.toLong)
      case rRat(n, _, d) => normalize(n.toLong, d.toLong)
      case rRat(n) => Rational(n.toLong)
      case rDec(w, _, f, null) => Rational(BigDecimal.apply(w + f))
      // FIXME implement properly the case where the fourth
  component is "eN"
      case rDec(w, _, f, e) => println(s"$w$f$e"); val b =
  BigDecimal.apply(w + f + e); println(s"$b"); Rational(b)
      case _ => throw new RationalException(s"invalid rational
  expression: $x")
    }
  }
  ```
  ```

Example: Rating

```
object Rating {
  val rRating = """^(\w*)(-(\d\d))?$""".r
  /**
    * Alternative apply method for the Rating class such that a single
String is decoded
    *
    * @param s a String made up of a code, optionally followed by a
dash and a number, e.g. "R" or "PG-13"
    * @return a Rating
    */
   def apply(s: String): Rating
  s match {
    case rRating(code, _, null) => apply(code, None)
    case rRating(code, _, age) => apply(code, Try(age.toInt).toOption)
    case _ => throw new Exception(s"parse error in Rating: $s")
  }
}
```

Parsing (2)

- Matching on regular expressions works pretty well…
  - but the method isn't the easiest to use and such parsers don't <u>compose</u> very well.

Parsing (2a)

- Let's try a more <u>functional</u> parser:

  That's to say, we extend ***Function1[S,T]***

  ```
  trait Parser[-S,+T] extends (S => T)
  ```

  - This would work fine: it takes input of type *S* and returns a result of type *T.* But what if we want to combine this parser with another which takes whatever input is left over and then returns something of type *U*?
  - We're going to need something that returns not a *T* but a tuple of *S* and *T*. Or we could define a trait *ParseResult[S,T]*. Then, from this result, we could get both our *T* value and the rest of the input. What do we need for that?

  ```
  trait Parser[S,+T] extends (S => ParseResult[S,T])
  trait ParseResult[S,+T]
  case class Success[S,+T](result: T, nextInput: S) extends ParseResult[S,T]
  case class Failure[S,+T](message: String, nextInput: S) extends ParseResult[S,T]
  ```

- We could write our own parser that way. In fact, the Scala classes are similar but not quite the same: *Parser* takes only one parametric type *T* because input is defined via an abstract *type* defined in *Parser*.

Parsing (2b)

- We're going to need some new compound types:
  - We will need to be able to represent the following types in our Parser:
    - *T1 followed by T2*—we could simply use *(T1,T2)* but Scala defines a type constructor ~ so we can write *T1~T2*.

This is really just a case class

    - *T1 otherwise T2*—in other words, alternation: if we can parse the input as a *T1*, that's what we get, otherwise we try to parse it as a *T2*.
    - *Maybe T*, that's to say 0 or 1 *T*s, equivalent to *Option[T]*.
    - *Sequence of T*, that's to say any number of *T*s (including zero), equivalent to *Seq[T]*.

Parsing (3)

- OK, now we just need to be able to define the grammar that our parser can operate on:
  - Take a look at this set of “productions” in BNF (Backus-Naur form) followed by examples:

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

  - `1+4.5-3` is an *expr*; `2*3.14/5` is a *term;* `3.1415927` is a *factor*; `(7-5)` is also a *factor*.
  - The **Scala Parser Combinator** library allows us to code this parser with only a few substitutions:

```
import scala.util.parsing.combinator._
class Arith extends JavaTokenParsers {
  def expr: Parser[Any] = term~rep("+"~term | “-"~term);
  def term: Parser[Any] = factor~rep("*"~factor | “/"~factor);
  def factor: Parser[Any] = floatingPointNumber | “("~expr~")";
}
```

“~” replaces “ “; “rep(“ replaces “{“; “)” replaces “}”; “;” replaces “.” [although those “;” are entirely optional]

each method defines a *Parser[Any]*

Parsing (4)

- Let's try it in the REPL:

```
scala> val p = new Arith
p: Arith = Arith@78291b30
scala> val x = p.parseAll(p.expr,"1")
x: p.ParseResult[Any] = [1.2] parsed: ((1~List())~List())
```

We want to apply, specifically, the *expr* parser to "1"

TMI: Not very helpful output but it can be useful when debugging!

consumed text up to line 1, column 2

- That's not quite what we want!
  - We can get the result's "value":

```
scala> x.get
res1: Any = ((1~List())~List())
```

  - But that's not super useful either. For a start, it's an "Any" and secondly, what we've got is the concatenation of all the intermediate parse results. But, for now, we're not so interested in the internal workings of the parser.
  - So, how can we get the value "1" out of this?

- First, we need to understand how the *Parser* operators work:
  - https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html will take you to the root package
  - From there, you can click on *Parsers* to find:

```
p1 ~ p2 // sequencing: must match p1 followed by p2
p1 | p2 // alternation: must match either p1 or p2, with preference given to p1
p1.?    // optionality: may match p1 or not
p1.*    // repetition: matches any number of repetitions of p1
```

  - Now, you can understand what the parsers we defined before do:

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

***floatingPointNumber* is itself a Parser, defined in JavaTokenParsers, https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html**

  - There are many methods defined in *Parsers*, for example *rep*.

Parsing (6)

- These operators/methods work as follows:
  - Any (constant) string returns itself (as a *String*)
  - Any regular expression parser similarly returns the matched string(s)
  - A sequential composition *P~Q* returns both *P* and *Q*. This returns a "tilde" class written *[P~Q]* or, if you prefer, *~[P,Q]*
  - An alternation *P|Q* returns either *P* or *Q* but preferably *P*
  - A repetition *rep(P)* or *repsep(P, separator)* returns a *List[P]*
  - An option *opt(P)* returns an *Option[P]*

Parsing (7)

- We’re getting close but not quite there yet…
  - *Parser* defines the ^^ operator such that a parser definition of the form *P* ^^ *f* parses the input just like *P* (yielding result *R*) but the result of the ^^ operator is actually *f(R).*
  - For example:
    - `floatingPointNumber ^^ (_.toDouble)`

Does it bother you that ^^ seems to work just like *map*? It bothered me! Then I found that ^^ actually invokes *map* but also records a name.

- Now we’re ready to implement our arithmetic parser…

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

Parsing (8)

```
package edu.neu.coe.scala.parse
import scala.util.parsing.combinator._
/**
 * @author scalaprof
 */
class Arith extends JavaTokenParsers {
  trait Expression {
    def eval: Double
  }
  abstract class Factor extends Expression
  case class Expr(t: Term, ts: List[String~Term]) extends Expression {
    def term(t: String~Term): Double = t match {case "+"~x => x.eval; case "-"~x => -x.eval }
    def eval = ts.foldLeft(t.eval)(_ + term(_))
  }
  case class Term(f: Factor, fs: List[String~Factor]) extends Expression {
    def factor(t: String~Factor): Double = t match {case "*"~x => x.eval; case "/"~x => 1/x.eval }
    def eval = fs.foldLeft(f.eval)(_ * factor(_))
  }
  case class FloatingPoint(x: Any) extends Factor {
    def eval = x.toString.toDouble
  }
  case class Parentheses(e: Expr) extends Factor {
    def eval = e.eval
  }
  def expr: Parser[Expr] = term~rep("+"~term | "-"~term) ^^ { case t~r => r match {case x: List[String~Term] => Expr(t,x)}}
  def term: Parser[Term] = factor~rep("*"~factor | "/"~factor) ^^ { case f~r => r match {case x: List[String~Factor] =>
Term(f,x)}}
  def factor: Parser[Factor] = (floatingPointNumber | "("~expr~")") ^^ { case "("~e~")" => e match {case x: Expr =>
Parentheses(x)}; case s => FloatingPoint(s) }
}
```

Parsing (9)

```
scala> import edu.neu.coe.scala.parse._
import edu.neu.coe.scala.parse._

scala> val parser = new Arith
parser: edu.neu.coe.scala.parse.Arith = edu.neu.coe.scala.parse.Arith@48326b9d

scala> parser.parseAll(parser.expr, "1").get.eval
res0: Double = 1.0

scala> parser.parseAll(parser.expr, "1*2+1-3/2").get.eval
res1: Double = 1.5

scala> parser.parseAll(parser.expr, "1*2+1-pi/2").get.eval
java.lang.RuntimeException: No result when parsing failed
  at scala.sys.package$.error(package.scala:27)
  at scala.util.parsing.combinator.Parsers$NoSuccess.get(Parsers.scala:176)
  at scala.util.parsing.combinator.Parsers$NoSuccess.get(Parsers.scala:162)
  ... 43 elided
```

Oops! throwing an exception isn't very nice—but that's expected when we invoke *get*.

- We can build in a little error handling to avoid this:

```
  def expr: Parser[Expr] = term~rep("+"~term | "-"~term | failure("expr")) ^^ { case t~r => r match {case x:
List[String~Term] => Expr(t,x)}}
  def term: Parser[Term] = factor~rep("*"~factor | "/"~factor | failure("term")) ^^ { case f~r => r match {case x:
List[String~Factor] => Term(f,x)}}
  def factor: Parser[Factor] = (floatingPointNumber | "("~expr~")" | failure("factor")) ^^ { case "("~e~")" => e
match {case x: Expr => Parentheses(x)}; case s => FloatingPoint(s) }

scala> parser.parseAll(parser.expr, "1*2+1-pi/2")
res1: parser.ParseResult[parser.Expr] =
[1.7] failure: factor

1*2+1-pi/2
      ^
```

Parsing (10)—Rational

```
trait RationalNumber { def value: Try[Rational] }
class RationalParser extends JavaTokenParsers {
  def parse(w: String): Try[RationalNumber] = parseAll(number, w) match {
    case Success(t, _) => scala.util.Success(t)
    case Failure(m, _) => scala.util.Failure(RationalParserException(m))
    case Error(m, _) => scala.util.Failure(RationalParserException(m))
  }
  case class WholeNumber(sign: Boolean, digits: String) extends RationalNumber {
    override def value: Try[Rational] = scala.util.Success(Rational(BigInt(digits)).applySign(sign))
  }
  object WholeNumber {
    val one: WholeNumber = WholeNumber(sign = false, "1")
  }
  case class RatioNumber(numerator: WholeNumber, denominator: WholeNumber) extends RationalNumber {
    override def value: Try[Rational] = for (n <- numerator.value; d <- denominator.value) yield n / d
  }
  case class RealNumber(sign: Boolean, integerPart: String, fractionalPart: String, exponent: Option[String]) extends
RationalNumber {
    override def value: Try[Rational] = {
      val bigInt = BigInt(integerPart + fractionalPart)
      val exp = exponent.getOrElse("0").toInt
      Try(Rational(bigInt).applySign(sign).applyExponent(exp - fractionalPart.length))
    }
  }
  def number: Parser[RationalNumber] = realNumber | ratioNumber
  def ratioNumber: Parser[RatioNumber] = simpleNumber ~ opt("/" ~> simpleNumber) ^^ { case n ~ maybeD => RatioNumber(n,
maybeD.getOrElse(WholeNumber.one)) }
  def simpleNumber: Parser[WholeNumber] = opt("-") ~ wholeNumber ^^ { case so ~ n => WholeNumber(so.isDefined, n) }
  def realNumber: Parser[RealNumber] = opt("-") ~ wholeNumber ~ ("." ~> wholeNumber) ~ opt(E ~> wholeNumber) ^^ { case so ~
integerPart ~ fractionalPart ~ expo => RealNumber(so.isDefined, integerPart, fractionalPart, expo) }
  private val E = "[eE]".r
}
object RationalParser {
  val parser = new RationalParser

  def parse(s: String): Try[Rational] = parser.parse(s).flatMap(_.value)
}
case class RationalParserException(m: String) extends Exception(m)
```

Parsing (wrap-up)

- Best sources of information for Parsing:
  - *Programming in Scala* (Odersky & Spoon)
  - Latest API docs
  - Code examples: http://booksites.artima.com/programming_in_scala_2ed/examples/html/ch33.html
  - A somewhat more practical document on this (though I say so myself): http://scalaprof.blogspot.com/2015/10/scalas-parser-combinators.html
  - And a rather more advanced parser problem written up here: https://www.javadoc.io/doc/org.scala-lang.modules/scala-parser-combinators_2.13/latest/scala/util/parsing/combinator/index.html
  - TableParser
  - Matchers