Parsing & Language Architecture
The evolution of a DSL?

We are here
Towards a language architecture
Grammars

A notation for defining all the syntactically valid programs of a language. (Whitespace usually ignored.)
Grammars (Is this a DSL?)

A notation for defining all the syntactically valid programs of a language. (Whitespace usually ignored.)

expr ::= 
  expr + expr
| expr - expr
| expr * expr
| expr / expr
| n
Parser combiners

An internal DSL for recursive-descent parsers

import scala.util.parsing.combinator._

object Parser extends JavaTokenParsers {
  def expr: Parser[String] =
    ( expr ~ "+" ~ expr |
      expr ~ "-" ~ expr |
      expr ~ "*" ~ expr |
      expr ~ "/" ~ expr |
      wholeNumber )
}

Warning: left-recursion

libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
Packrat parsing

Allows left-recursion. Recursive-descent parsing with backtracking.

```scala
import scala.util.parsing.combinator._
object Parser extends JavaTokenParsers with PackratParsers {

  lazy val expr: PackratParser[AST] =
  ( expr ~ "+" ~ expr |
    expr ~ "-" ~ expr |
    expr ~ "+" ~ expr |
    expr ~ "/" ~ expr |
    wholeNumber )
}
```

`build.sbt`

```scala
libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
```
Abstract syntax

Describes the intermediate representation, i.e., the abstract syntax tree. An inductive data structure.

\[
\begin{align*}
\text{n} & \in \mathbb{Z} \\
\text{expr} & ::= \\
\text{expr} & + \text{expr} \\
| & \text{expr} - \text{expr} \\
| & \text{expr} \ast \text{expr} \\
| & \text{expr} / \text{expr} \\
| & \text{n}
\end{align*}
\]

sealed abstract class Expr

case class Plus(left: Expr, right: Expr) extends Expr

case class Sub(left: Expr, right: Expr) extends Expr

case class Mult(left: Expr, right: Expr) extends Expr

case class Div(left: Expr, right: Expr) extends Expr

case class Num(n: Int) extends Expr
import scala.util.parsing.combinator._

object Parser extends JavaTokenParsers with PackratParsers {

  lazy val expr: PackratParser[String] =

    (      
      expr ~ "+" ~ expr
    | expr ~ "-" ~ expr
    | expr ~ "*" ~ expr
    | expr ~ "/" ~ expr
    | wholeNumber 
    )

}

Warning: associativity / precedence

build.sbt

libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
import scala.util.parsing.combinator._

object Parser extends JavaTokenParsers with PackratParsers {

  lazy val expr: PackratParser[AST] =

  (   expr ~ "+" ~ expr ^^ {case l~"+"~r ⇒ Plus(l,r) } 
  | expr ~ "-" ~ expr ^^ {case l~"-"~r ⇒ Minus(l,r) } 
  | expr ~ "*" ~ expr ^^ {case l~"*"~r ⇒ Times(l,r) } 
  | expr ~ "/" ~ expr ^^ {case l~"/"~r ⇒ Divide(l,r)} 
  | wholeNumber       ^^ {s ⇒ Num(s.toInt)} )

}

Warning: associativity / precedence

libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
A less ambiguous grammar

The “lower-down” the operation, the higher its precedence.

\[ n \in \mathbb{Z} \]

\[
\text{expr} ::= \\
\text{expr} + \text{term} | \text{expr} - \text{term} | \text{fact}
\]

\[
\text{term} ::= \\
\text{term} * \text{fact} | \text{term} / \text{fact} | \text{fact}
\]

\[
\text{fact} ::= \\
n | ( \text{expr} )
\]
A Scala architecture for languages

**Read-Eval-Print-Loop (REPL)**

```
libraryDependencies += "org.scala-lang" % "scala-compiler" % scalaVersion.value
```

**Tests**

```
libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.13.0" % "test"
libraryDependencies += "org.scalatest" %% "scalatest" % "2.2.6" % "test"
```

**Parser Combinators**

**Case Classes**

**Functions & Pattern Matching**
Let's practice!

With a grammar that fixes the associativity / precedence problems

---

**External DSLs**

**Running the initial version of the code**

You should be able to do `sbt run` to run an initial version of the calculator interpreter. You should also be able to do `sbt test` to run some auto-generated tests of the initial parser and interpreter.

**Working with ScalaIDE**

You should be able to do `sbt eclipse` to generate a ScalaIDE project. Then, you can import the project in the usual way. Once in ScalaIDE, you can run the interpreter by opening the file `src/main/scala/calculator/calc.scala` and running it.

**Running tests in ScalaIDE:** You can run the tests by opening a test file and running it. Some of the tests are written with the ScalaCheck testing library, which isn't integrated into Eclipse's test runner (the thing with the green bar). The output from these tests will appear in the console, instead.

**Extend the calculator language to add new features**

Extend the code to implement the following grammar:

\[
\begin{align*}
  n & \in \mathbb{Z} \\
  e & \in \text{Expr} ::= e + t \mid e - t \mid t \\
  t & \in \text{Term} ::= t * f \mid t / f \mid f \\
  f & \in \text{Fact} ::= n \mid (e)
\end{align*}
\]

It's best to add features in the following order: