Parsing & Language Architecture
Grammars

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

Grammars (Is this a DSL?)

expr ::= 
    expr + expr 
| expr − expr 
| expr * expr 
| expr / expr 
| n
Parser combinators
An internal DSL for recursive-descent parsers

```scala
import scala.util.parsing.combinator._

object Parser extends JavaTokenParsers {

  def expr: Parser[String] =
  (    expr ~ "+" ~ expr
|    expr ~ "-" ~ expr
|    expr ~ "*" ~ expr
|    expr ~ "/" ~ expr
|    wholeNumber )

}
```

build.sbt

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

Warning: left-recursion
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 )
}

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

Warning: associativity / precedence
Abstract syntax

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

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

abstract class AST

abstract class Expr extends AST

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

Corresponds to case classes (algebraic data types) in 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

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

Warning: associativity / precedence
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

build.sbt

libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.4"
Language Architecture in Scala
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/calculator.scala` and running it.

**Running tests in ScalaIDE:** The `ScalaCheck` testing library doesn’t seem to work well with Eclipse yet. You’ll probably want to run the tests outside of Eclipse, using `sbt`.

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

Extend the code to implement the following grammar:

```
  n ∈ Z
  e ∈ Expr ::= e + t | e - t | t
  t ∈ Term ::= t * f | t / f | f
  f ∈ Fact ::= n | ( e )
```

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