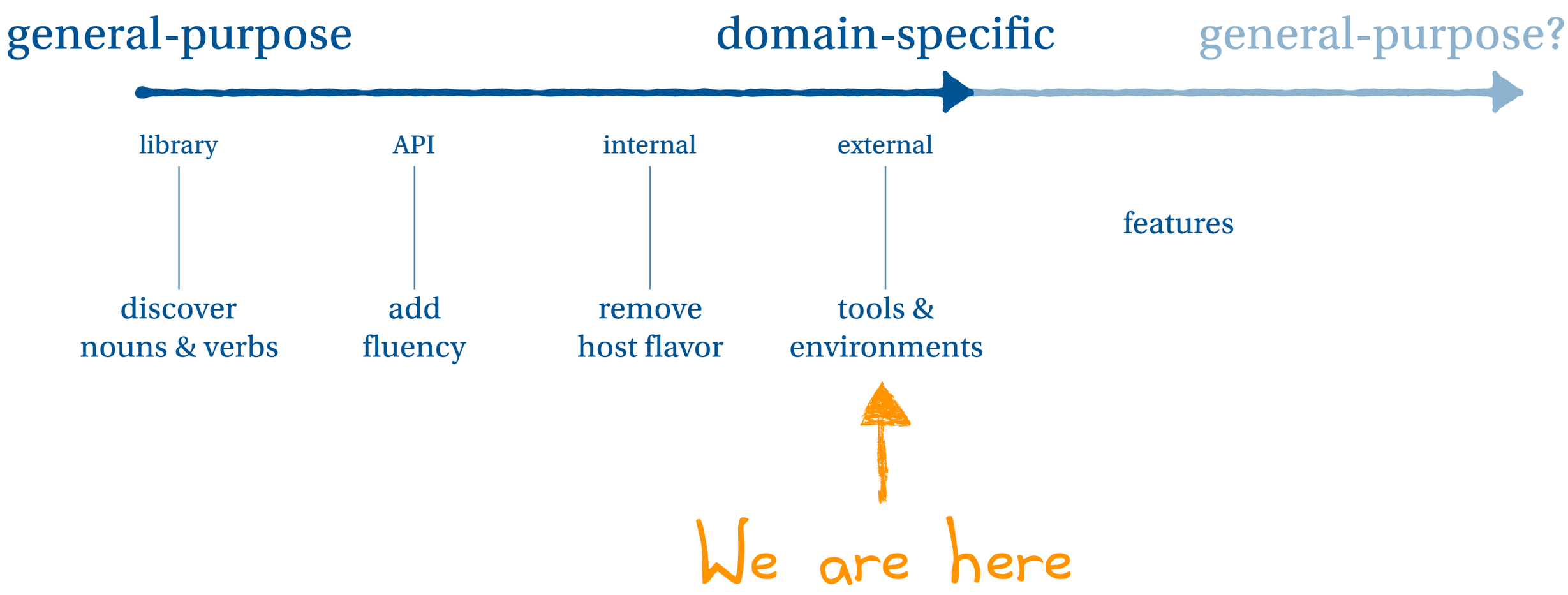
Parsing & Language Architecture

The evolution of a DSL?



Towards a language architecture

Grammars

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

start symbol

$n \in \mathbb{Z}$

```
expr ::=
        expr    +    expr
      | expr    -    expr
      | expr    *    expr
      | expr    /    expr
      | n
```

alternative

production rule

terminal

non-terminal

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 combinators

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

build.sbt

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

Packrat parsing

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

```
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 )

}
```

Warning: associativity / precedence

build.sbt

```
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.

$n \in \mathbb{Z}$

`expr ::=`	`sealed abstract class Expr`
`expr + expr`	`case class Plus(left: Expr, right: Expr) extends Expr`
`| expr - expr`	`case class Sub(left: Expr, right: Expr) extends Expr`
`| expr * expr`	`case class Mult(left: Expr, right: Expr) extends Expr`
`| expr / expr`	`case class Div(left: Expr, right: Expr) extends Expr`
`| n`	`case class Num(n: Int) extends Expr`

Actions: transform strings to IR

```
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"
```

Actions: transform strings to IR

```
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"
```

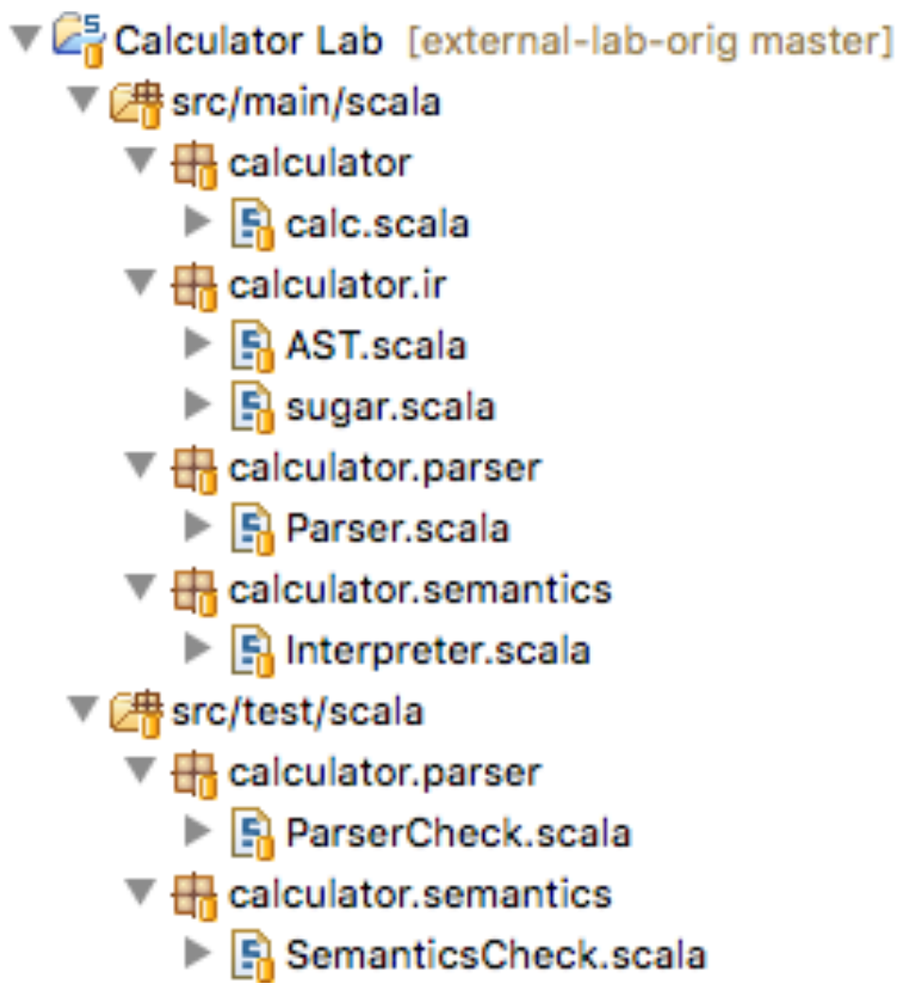
A less ambiguous grammar

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

```
                n ∈ ℤ
expr ::=
        expr    +    term
      | expr    -    term
      | fact

term ::=
        term    *    fact
      | term    /    fact
      | fact

fact ::=
        n | ( expr )
```

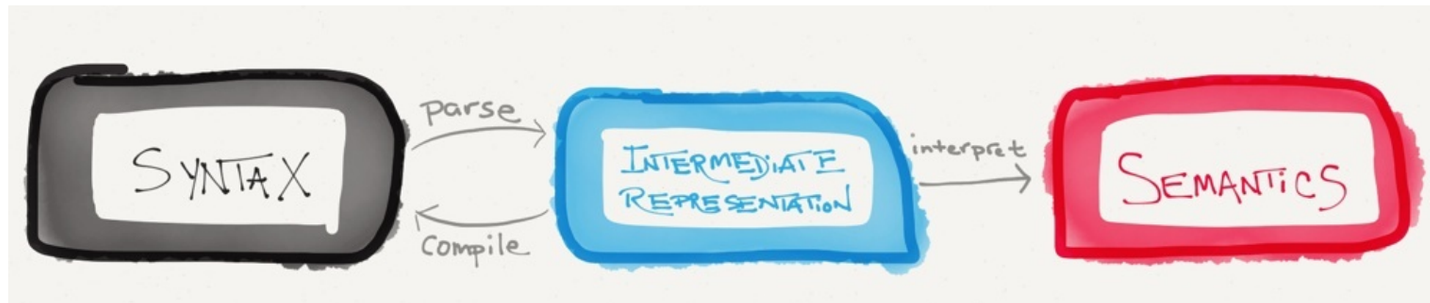
```scala
sealed abstract class Expr

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
```

A Scala architecture for languages



Read-Eval-Print-Loop (REPL)

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



parser combinators	case classes	functions & pattern matching

tests

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

Let’s practice!

With a grammar that fixes the associativity / precedence problems

.gitignore	Adding .gitignore	13 hours ago
README.md	initial commit	13 hours ago
build.sbt	initial commit	13 hours ago

README.md

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: 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:

Graphs

Settings

HTTPS clone URL

https://github.com

You can clone with HTTPS, SSH, or Subversion.

Clone in Desktop

Download ZIP