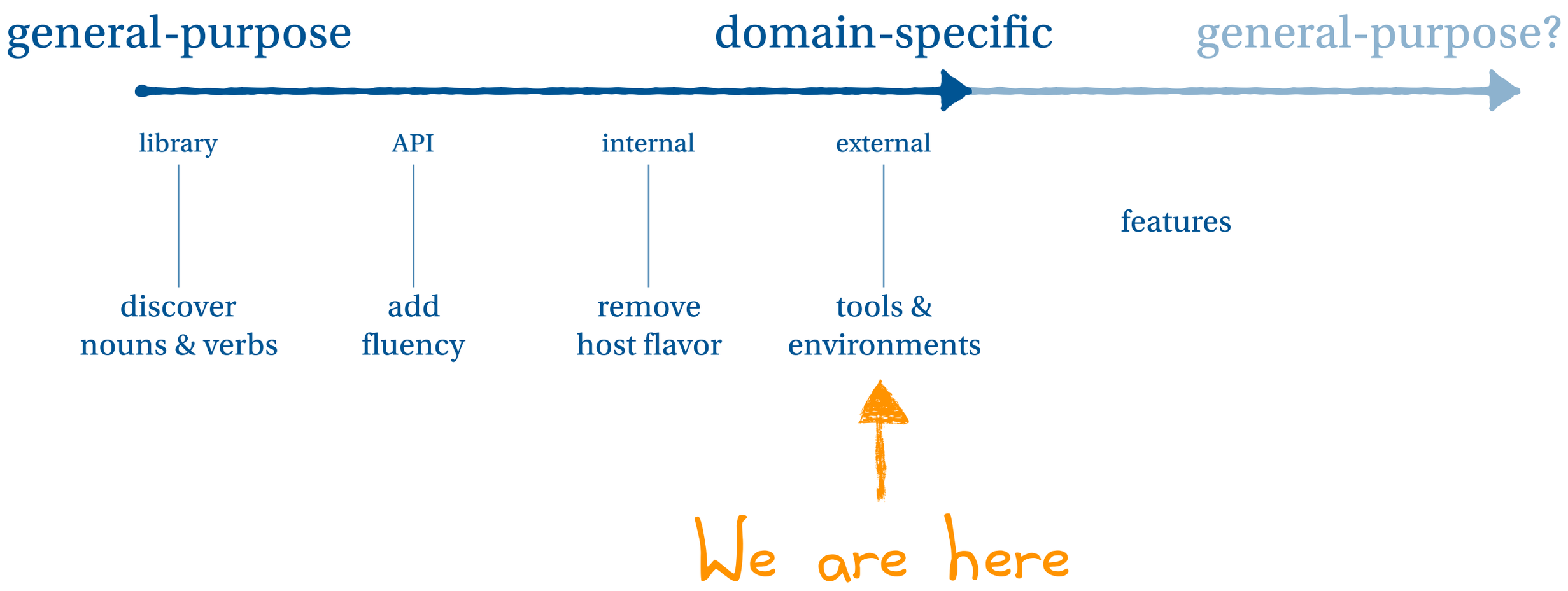
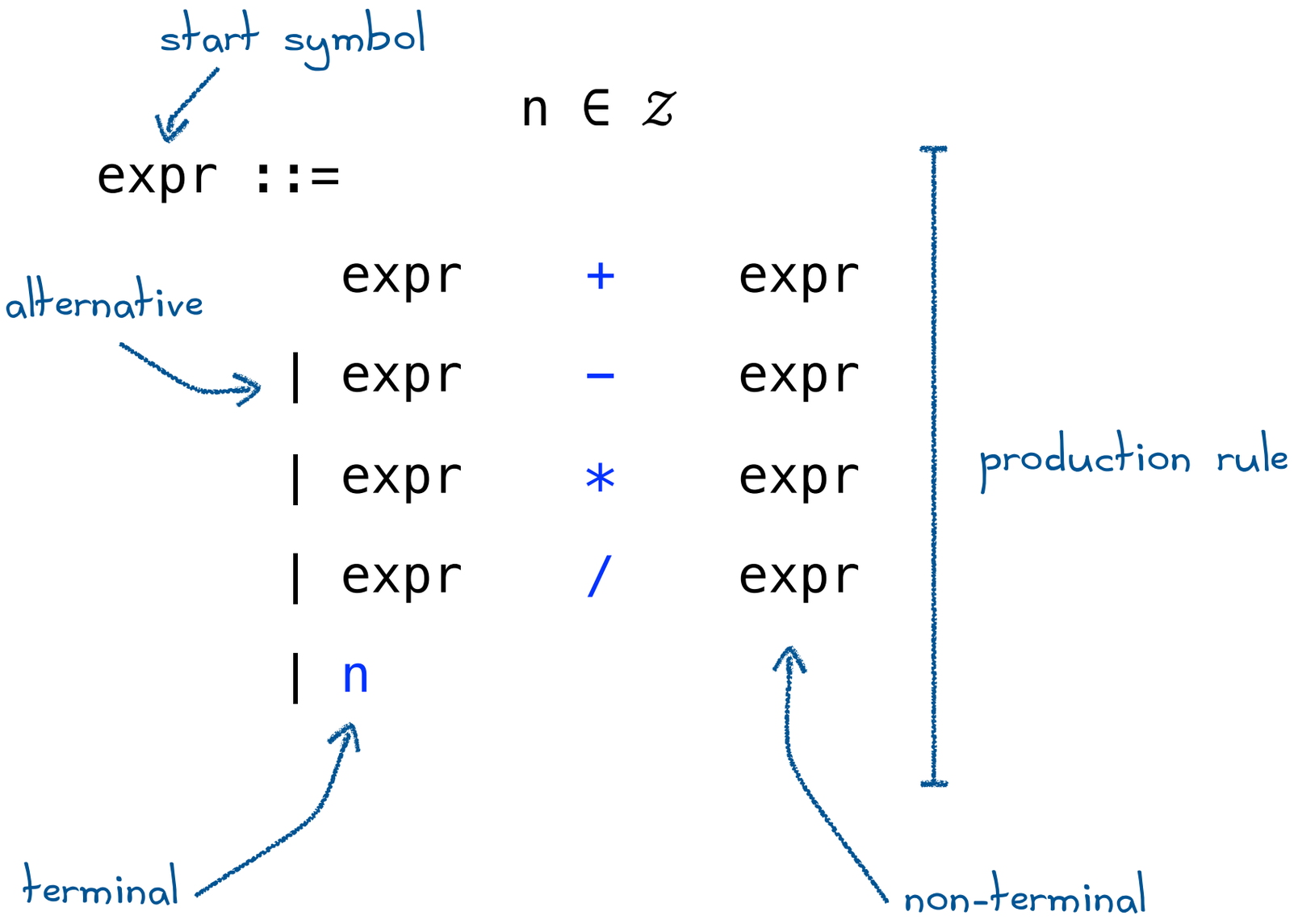
Parsing & Language Architecture

The evolution of a DSL?



Grammars

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



Abstract syntax

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

$$n \in \mathcal{Z}$$

expr ::=			**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

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[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[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[Expr] =

          (   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"
```

An ambiguous grammar

Precedence: If an expression has two *different* operations, which should be applied first?

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

A less ambiguous grammar

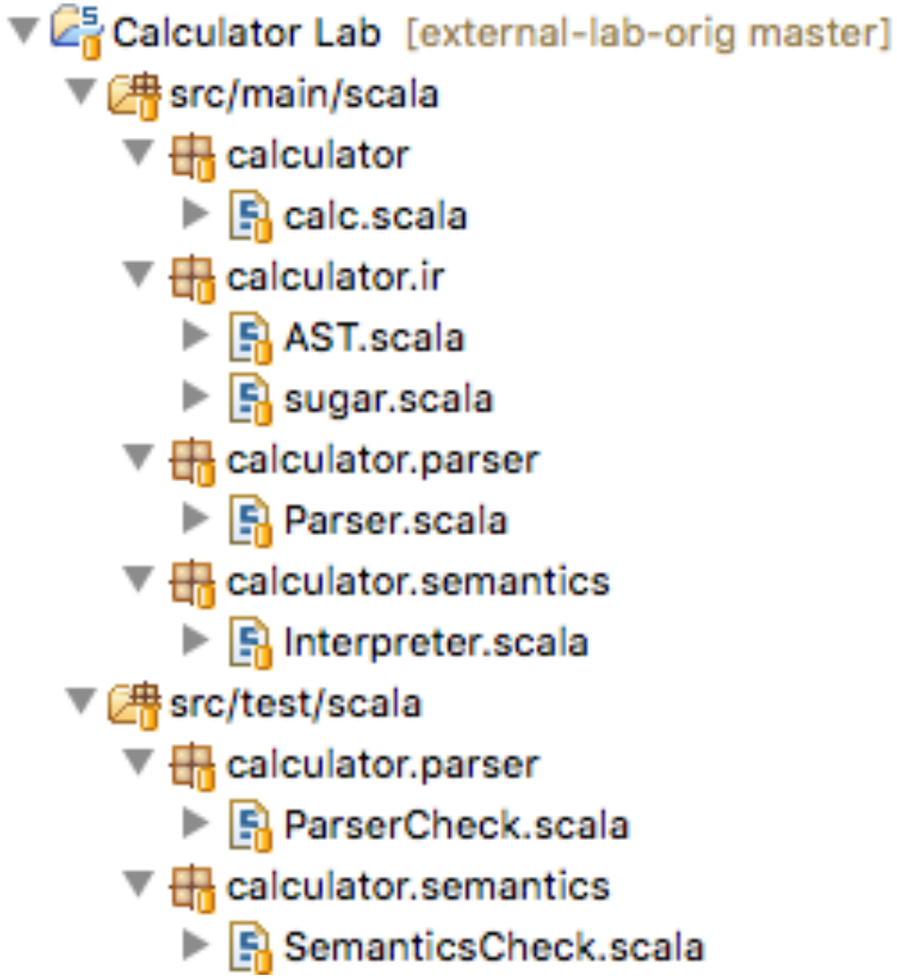
The “lower-down” the operation is in the grammar, the higher its precedence.

$$n \in \mathbb{Z}$$

```
expr ::=
        expr   +   term
      | expr   -   term
      | term

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

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

```
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"
```