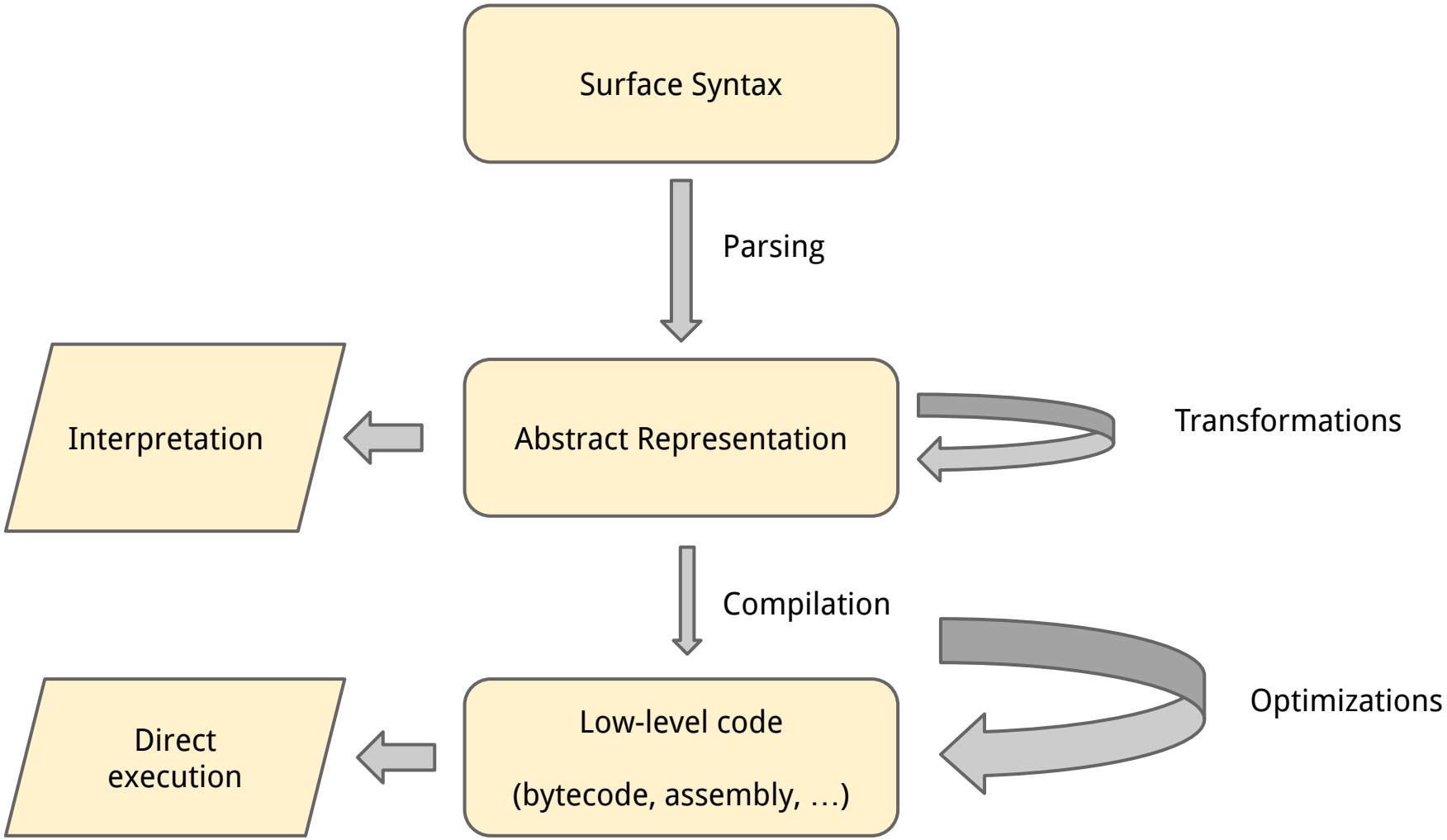
Introduction to Interpretation

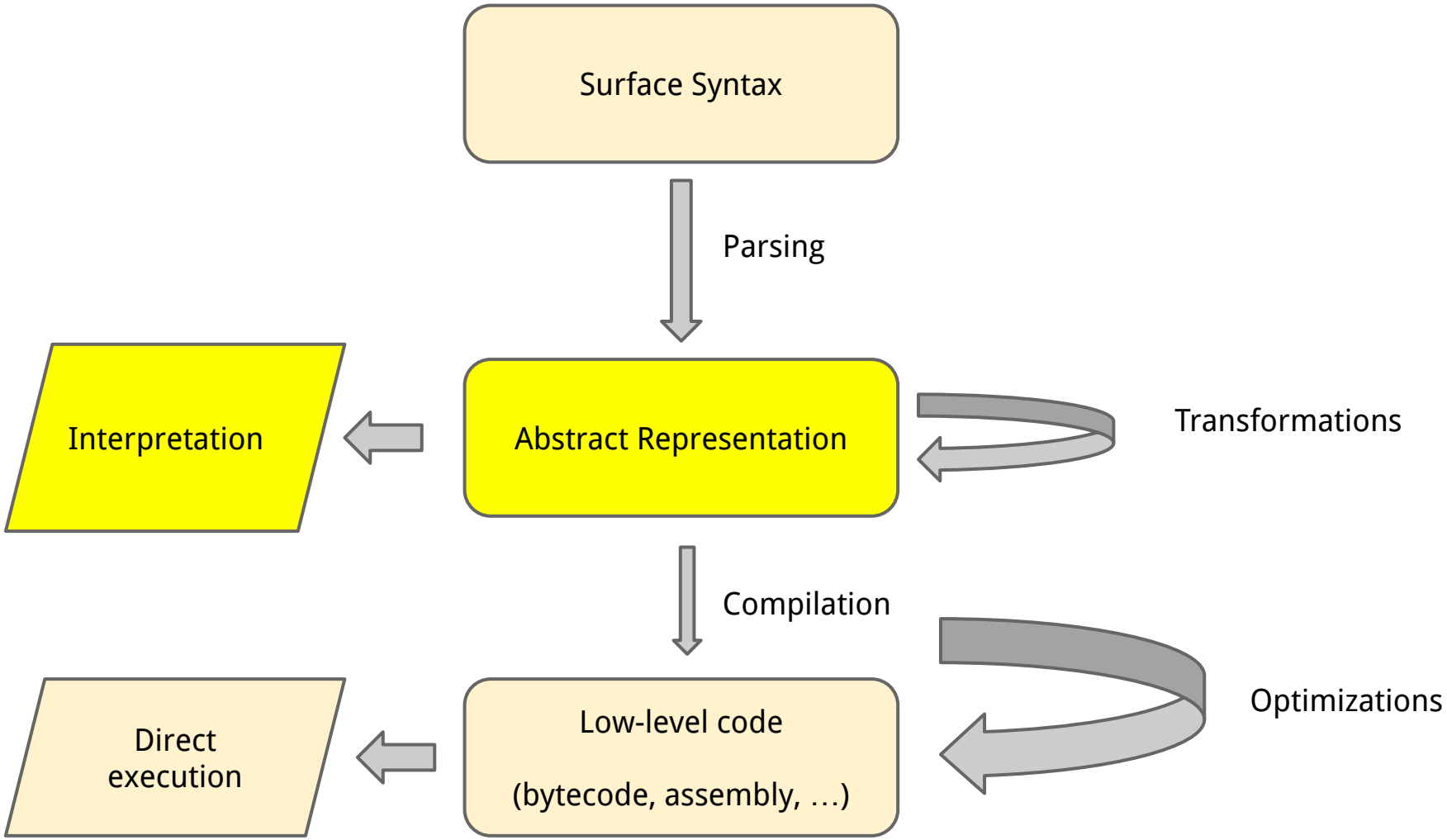
January 23, 2020

Riccardo Pucella

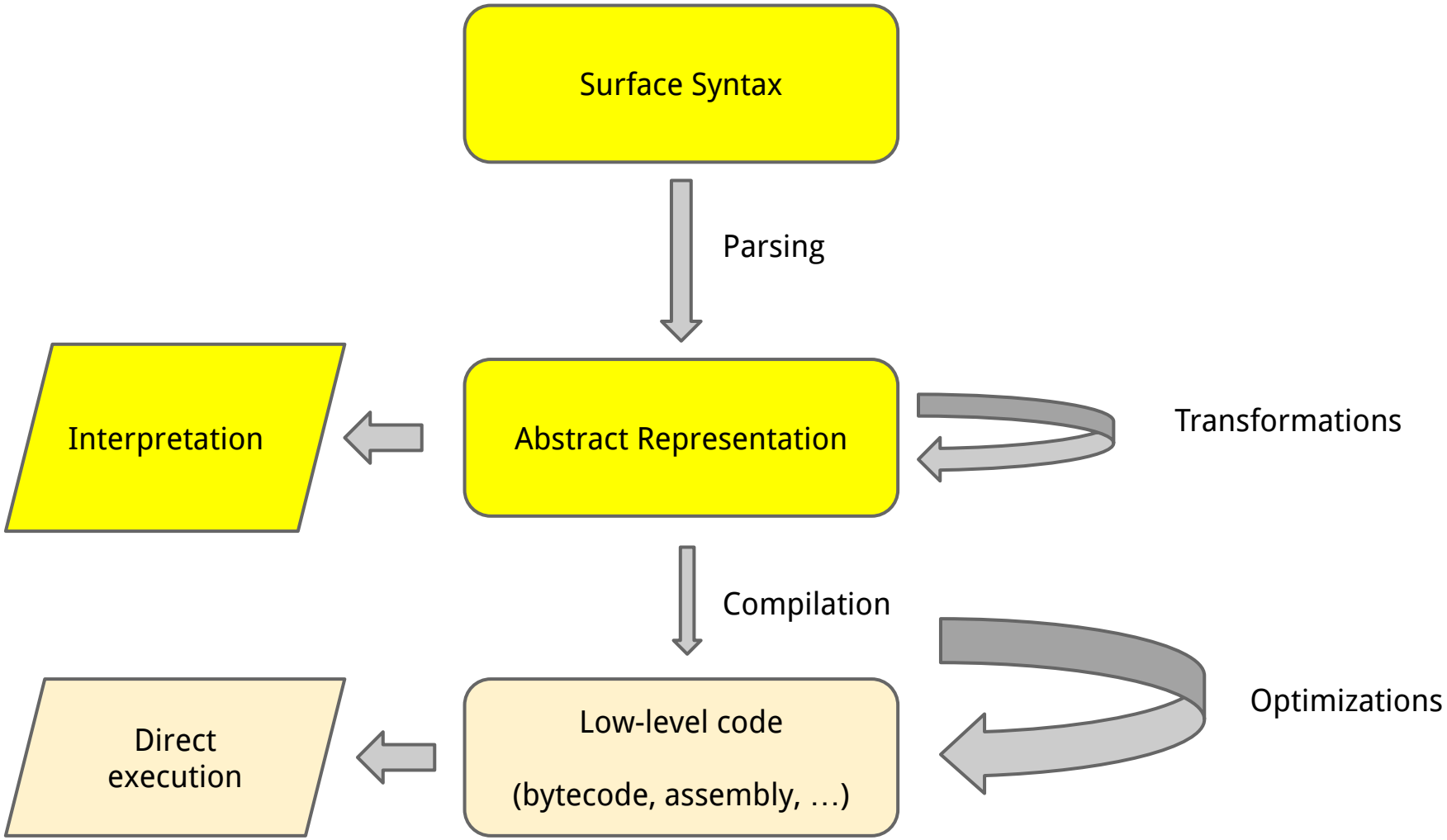
The structure of language execution



The structure of language execution



The structure of language execution



A simple expression language

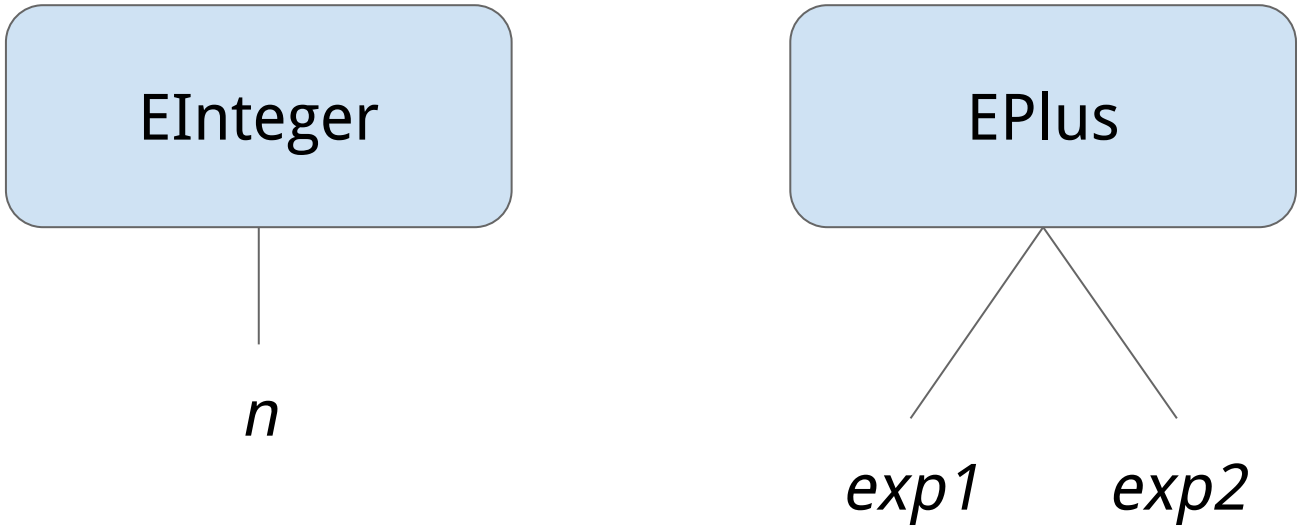
- We’re going to build-up a small language of mathematical expressions

- Computing over the integers
  - Operations +, *
  - Easy to expand (-, mod, div, …)

- Abstract representation needs to account for nesting expressions
  - E.g., (3 + 4) * (5 + 6)

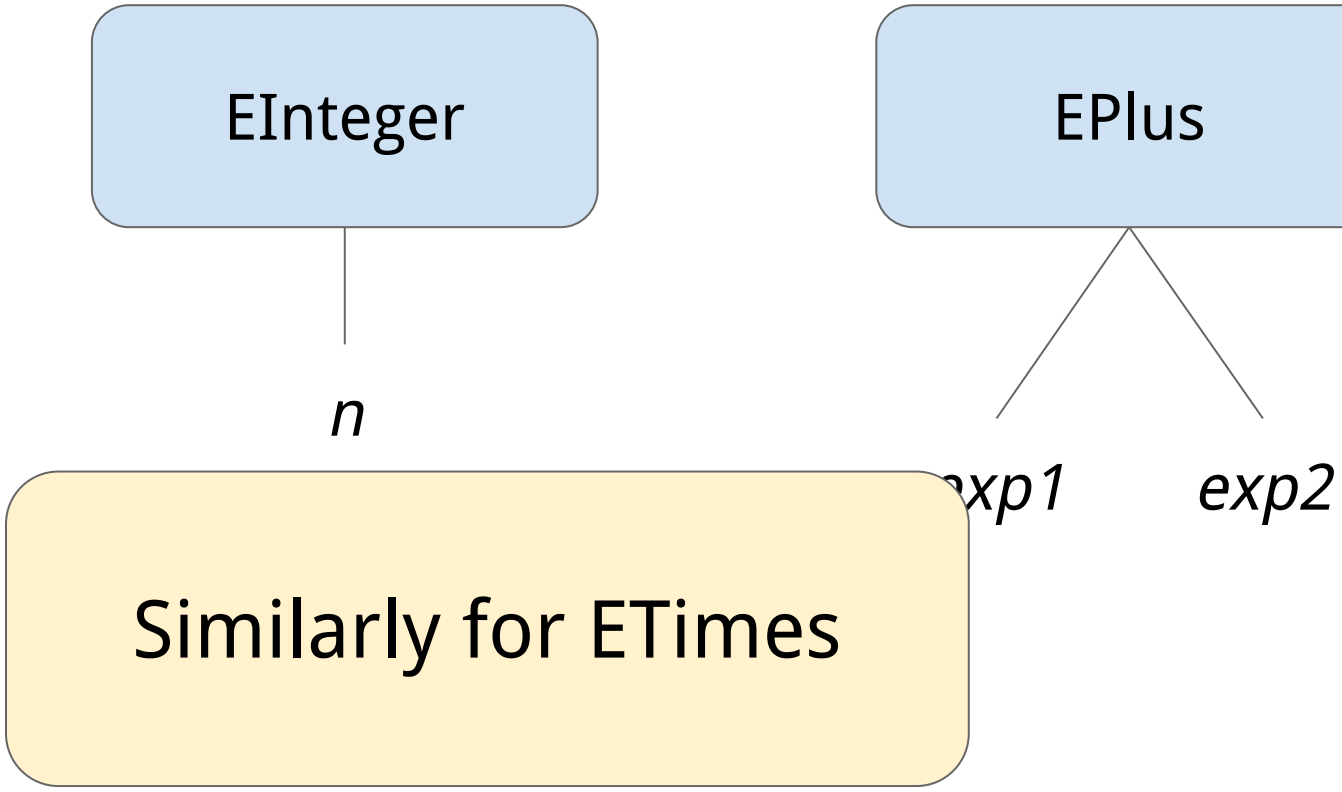
Abstract representation

An expression is a tree. Nodes are kinds of expressions:

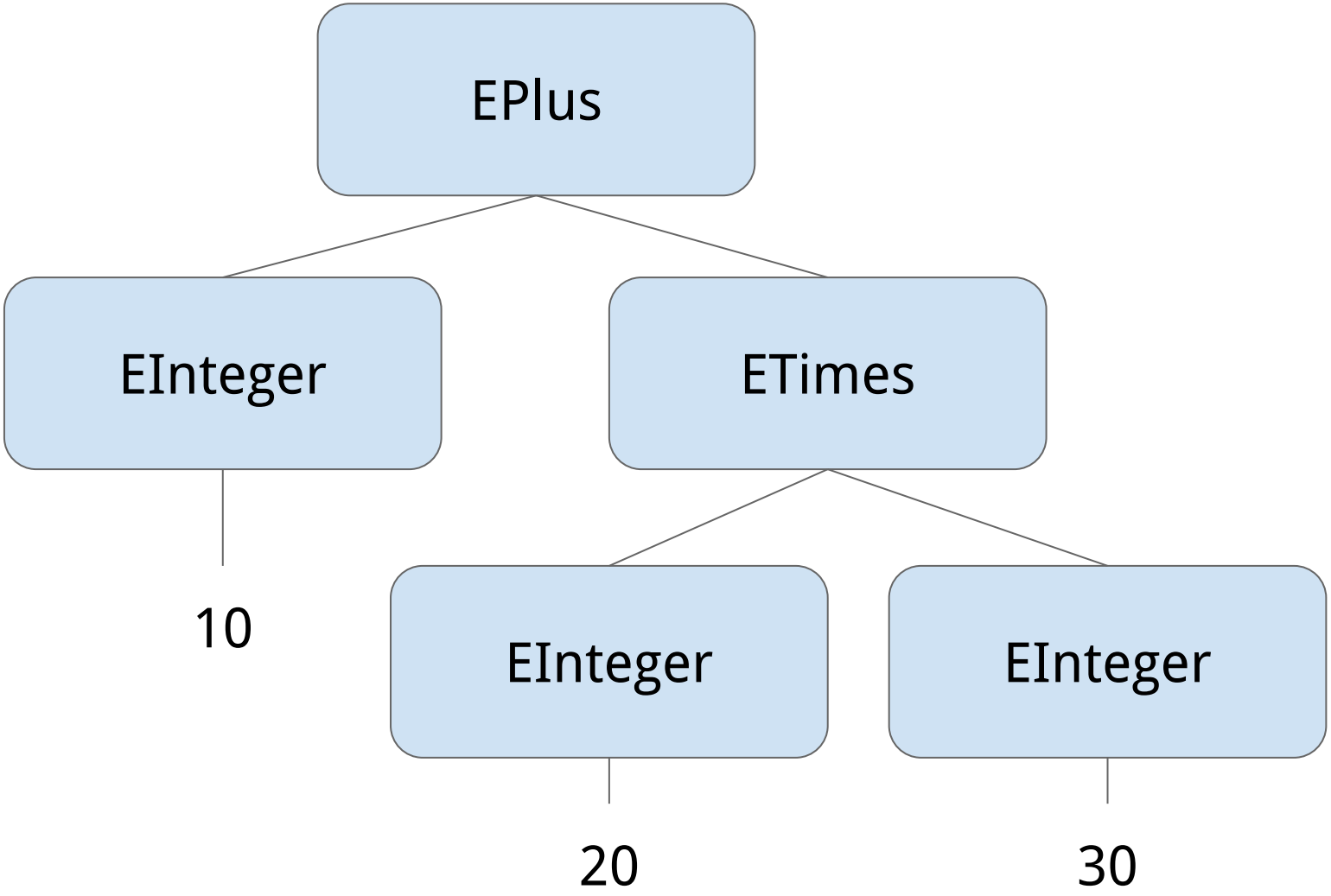


Abstract representation

An expression is a tree. Nodes are kinds of expressions:



Example: 10 + (20 * 30)



Abstract Representation in Scala

```
abstract class Exp

class EInteger (val i:Int) extends Exp {
   override def toString () : String =
      "EInteger(" + i + ")"

class EPlus (val e1:Exp, val e2:Exp) extends Exp {
    override def toString () : String =
      "EPlus(" + e1 + "," + e2 + ")"

class ETimes (val e1:Exp, val e2:Exp) extends Exp {
    override def toString () : String =
      "ETimes(" + e1 + "," + e2 + ")"
```

Abstract Representation in Scala

```
abstract class Exp

class EInteger (val i:Int) extends Exp {
```

Constructing our example:

```
new EPlus(new EInteger(10),
          new ETimes(new EInteger(20),
                     new EInteger(30)))
```

Evaluation

Evaluation is the process of taking an expression and reducing it to a value

Every node in the abstract representation has an evaluation method that evaluates the expression to a value

- evaluating an expression requires recursively evaluating subexpressions

Semantics?

Evaluation

```
abstract class Exp {

    def eval (): Int

}
```

Evaluation for integer literals

```
class EInteger (val i:Int) extends Exp {

    override def toString () : String =
       "EInteger(" + i + ")"

    def eval () : Int =
       i




}
```

Evaluation for addition

```
class EPlus (val e1:Exp, val e2:Exp) extends Exp {

    override def toString () : String =
       "EPlus(" + e1 + "," + e2 + ")"

    def eval () : Int = {
       val i1 = e1.eval()
       val i2 = e2.eval()
       return i1 + i2
    }

}
```

Evaluation for multiplication

```
class ETimes (val e1:Exp, val e2:Exp) extends Exp {

    override def toString () : String =
       "ETimes(" + e1 + "," + e2 + ")"

    def eval () : Int = {
       val i1 = e1.eval()
       val i2 = e2.eval()
       return i1 * i2
    }

}
```

Conditionals

Let's make evaluation depends on a condition

In C / Javascript, you have expression

```
cond ? then : else
```

Evaluates to *then* if *cond* is true, *else* otherwise

Take a condition to be true if it is non-zero

- Next time, we add actual Booleans

Conditionals

```
class EIf (val c:Exp, val t:Exp, val e:Exp) extends Exp {

    override def toString () : String =
       "EIf(" + c + "," + t + "," + e + ")"

    def eval () : Int = {
       val ci = c.eval()
       if (ci == 0) {
          return e.eval()
       } else {
          return t.eval()
       }
    }
}
```