“Going bananas with recursion schemes for fixed point data types”

Pawel Szulc

email: paul.szulc@gmail.com
twitter: @rabbitonweb
github: https://github.com/rabbitonweb/

Software Engineering

Software engineering

*“In late **1967** the Study Group recommended the holding of a working **conference** on **Software Engineering**. The phrase ‘**software engineering**’ was deliberately **chosen** as being **provocative**, in **implying the need** for software manufacture to be based on the types of **theoretical foundations** and **practical disciplines**, that are traditional in the established branches of engineering.” [NATO68]*

Elegance of Functional Programming

*"Programming notation can be expressed by "**formulæ** and **equations** (...) which share the **elegance** of those which underlie **physics** and **chemistry** or any other **branch** of basic **science**". -- [BdM97]*

Recursion

“Recursion is where Functional Programming hits the bottom”

- Greg Pfeil

Recursion Schemes

“Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire” by Erik Meijer Maarten, Maarten Fokkinga and Ross Paterson *[MFP91]*

Recursion Schemes

“Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire” by Erik Meijer Maarten, Maarten Fokkinga and Ross Paterson *[MFP91]*

Describes:

- Simple, composable combinators (**recursion schemes**)
- Process of traversing **recursive structures**

Real world application

https://github.com/quasar-analytics/quasar

https://github.com/slamdata/matryoshka

Recursive Structures are common

Recursive Structures are common

- Simplest examples
  - List:
    - Cons(a, List)
    - Nil
  - Binary Tree
    - Node(a, Tree, Tree)
    - Leaf(a)

Recursive Structures are common

- Simplest examples
    - List:
        - Cons(a, List)
        - Nil
    - Binary Tree
        - Node(a, Tree, Tree)
        - Leaf(a)
- To name few examples:
    - File systems
    - 3D Graphics
    - Databases

Recursive Structures are common

- Simplest examples
    - List:
        - Cons(a, List)
        - Nil
    - Binary Tree
        - Node(a, Tree, Tree)
        - Leaf(a)
- To name few examples:
    - File systems
    - 3D Graphics
    - Databases
- Any nested, inductively defined type

Expressions!

Our example

$$10^2 + 20.5 * 14.5$$

Expressions!

Our example

$$10^2 + 20.5 * 14.5$$

```
Sum (
    Square( IntValue(10) )
    Multiply( DecValue(20.5),
DecValue(14.5) ),
)
```

Expressions!

Our example

$$10^2 + 20.5 * 14.5$$

Sum (

Square(IntValue(10))

Multiply(DecValue(20.5),

DecValue(14.5)),

)

Expressions!

Our example

$10^2 + 20.5 * 14.5$

```
Sum (
      Square( IntValue(10) )
      Multiply( DecValue(20.5),
DecValue(14.5) ),
 )
```

Expressions!

Our example

$$10^2 + 20.5 * 14.5$$

```
Sum (
    Square( IntValue(10) )
    Multiply( DecValue(20.5),
DecValue(14.5) ),
 )
```

Expressions!

Our example

$10^2 + 20.5 * 14.5$

```
Sum (
    Square( IntValue(10) )
    Multiply( DecValue(20.5),
DecValue(14.5) ),
)
```

Expressions!

Our example

$10^2 + \mathbf{20.5} * \mathbf{14.5}$

Sum (

Square(IntValue(10))

Multiply(**DecValue(20.5)**,

DecValue(14.5)),

Expressions!

Our example

$10^2 + 20.5 * 14.5$

```
Sum (
    Square( IntValue(10) )
    Multiply( DecValue(20.5),
DecValue(14.5) ),
 )
```

Expressions!

Our example

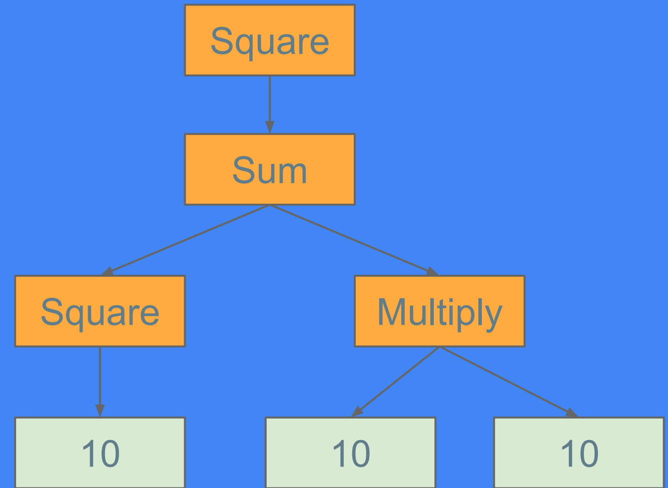
$$(10^2 + 20.5 * 14.5)^2$$

```
Square (
    Sum (
        Square( IntValue(10) )
        Multiply( DecValue(20.5),
DecValue(14.5) ),
    )
)
```

Expressions!

Our example

$(10^2 + 20.5 * 14.5)^2$



```
data Exp = IntVal Int          |
           DecVal Double       |
           Sum Exp Exp         |
           Multiply Exp Exp    |
           Divide Exp Exp      |
           Square Exp
```

```
data Exp = IntVal Int          |
           DecVal Double       |
           Sum Exp Exp         |
           Multiply Exp Exp    |
           Divide Exp Exp      |
           Square Exp
```

```
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```
data Exp = IntVal Int           |
           DecVal Double        |
           Sum Exp Exp          |
           Multiply Exp Exp     |
           Divide Exp Exp       |
           Square Exp
```

```
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val evaluate: Exp => Double =
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)              => v.toDouble
  case DecValue(v)              => v
  case Sum(exp1, exp2)          => evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) => evaluate(exp1) * evaluate(exp2)
  case Square(exp)              =>
                                     val v = evaluate(exp)
                                     v * v
  case Divide(exp1, exp2)     => evaluate(exp1) / evaluate(exp2)
}
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val mkString: Exp => String = {
  case IntValue(v)          => v.toString
  case DecValue(v)          => v.toString
  case Sum(exp1, exp2)      => s"( ${mkStr(exp1)} + ${mkStr(exp2)})"
  case Multiply(exp1, exp2) => s"( ${mkStr(exp1)} * ${mkStr(exp2)})"
  case Square(exp)          => s"(${mkStr(exp)})^2"
  case Divide(exp1, exp2)   => s"( ${mkStr(exp1)} / ${mkStr(exp2)})"
}
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
             if(exp1 == exp2) => Square(optimize(exp1))
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
             if(exp1 == exp2) => Square(optimize(exp1))
 case IntValue(v)               => IntValue(v)
 case DecValue(v)               => DecValue(v)
 case Sum(exp1, exp2)           => Sum(optimize(exp1), optimize(exp2))
 case Multiply(exp1, exp2)      => Multiply(optimize(exp1), optimize(exp2))
 case Square(exp)               => Square(optimize(exp))
 case Divide(exp1, exp2)        => Divide(optimize(exp1), optimize(exp2))
}
```

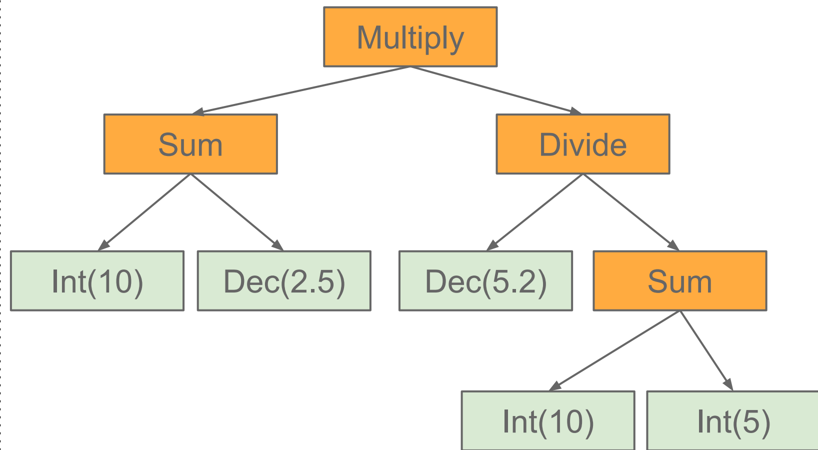
```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
             if(exp1 == exp2) => Square(optimize(exp1))
 case IntValue(v)             => IntValue(v)
 case DecValue(v)             => DecValue(v)
 case Sum(exp1, exp2)         => Sum(optimize(exp1), optimize(exp2))
 case Multiply(exp1, exp2)    => Multiply(exp1, optimize(exp2))
 case Square(exp)             => Square(optimize(exp))
 case Divide(exp1, exp2)      => Divide(optimize(exp1), optimize(exp2))
}
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

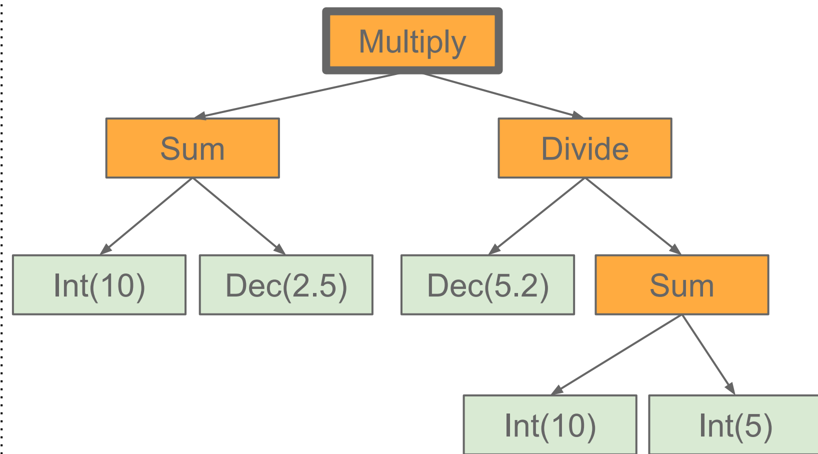
```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
             if(exp1 == exp2) => Square(optimize(exp1))
 case IntValue(v)             => optimize(IntValue(v))
 case DecValue(v)             => optimize(DecValue(v))
 case Sum(exp1, exp2)         => Sum(optimize(exp1), optimize(exp2))
 case Multiply(exp1, exp2)    => Multiply(optimize(exp1), optimize(exp2))
 case Square(exp)             => Square(optimize(exp))
 case Divide(exp1, exp2)      => Divide(optimize(exp1), optimize(exp2))
}
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



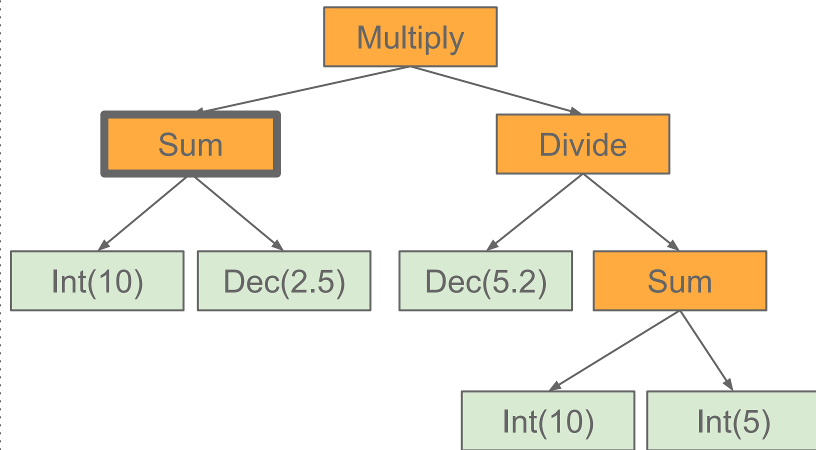

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)            => v.toDouble
  case DecValue(v)            => v
  case Sum(exp1, exp2)        =>
                evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
                evaluate(exp1) * evaluate(exp2)
  case Square(exp)            =>
                 val v = evaluate(exp)
                v * v
  case Divide(exp1, exp2)    =>
                evaluate(exp1) / evaluate(exp2)
}
```



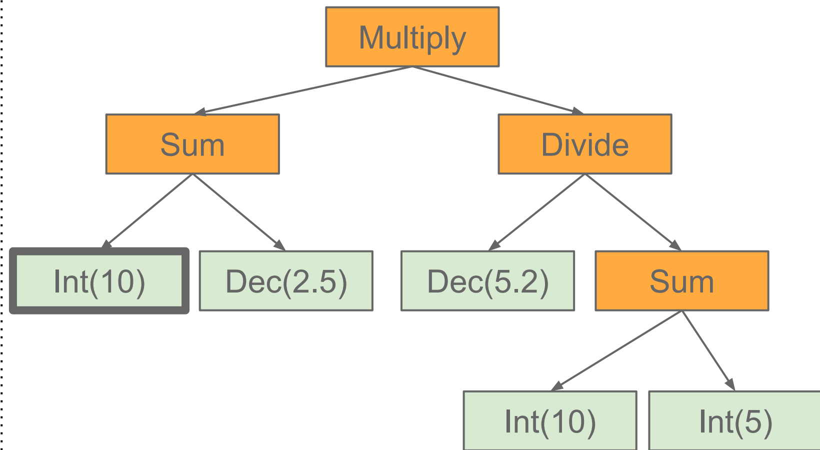

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)             => v.toDouble
  case DecValue(v)             => v
  case Sum(exp1, exp2)         =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)             =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



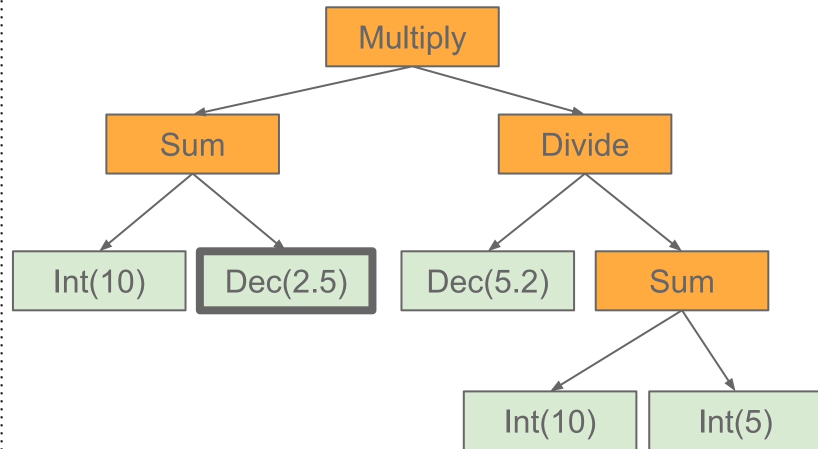

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)            => v.toDouble
  case DecValue(v)            => v
  case Sum(exp1, exp2)        =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)            =>
               val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



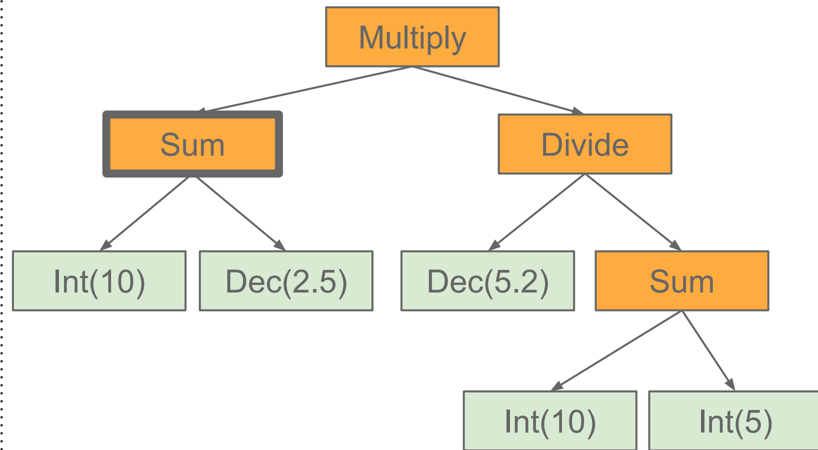

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
               val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



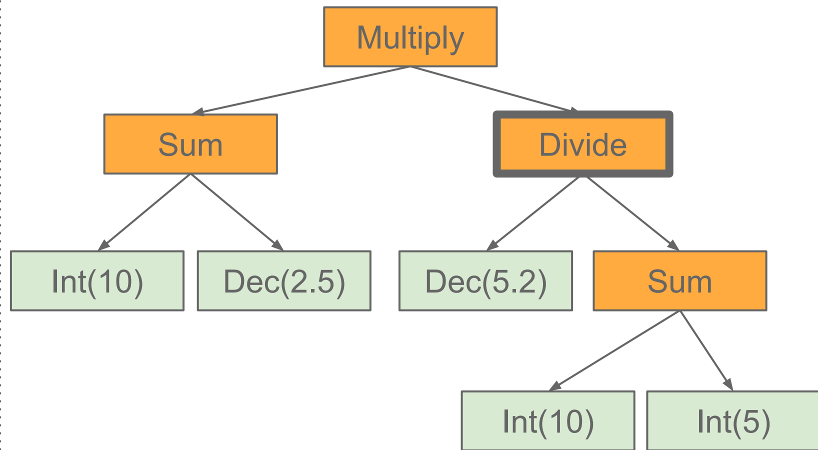

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
                 evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
                 evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
                  val v = evaluate(exp)
                 v * v
  case Divide(exp1, exp2)    =>
                 evaluate(exp1) / evaluate(exp2)
}
```



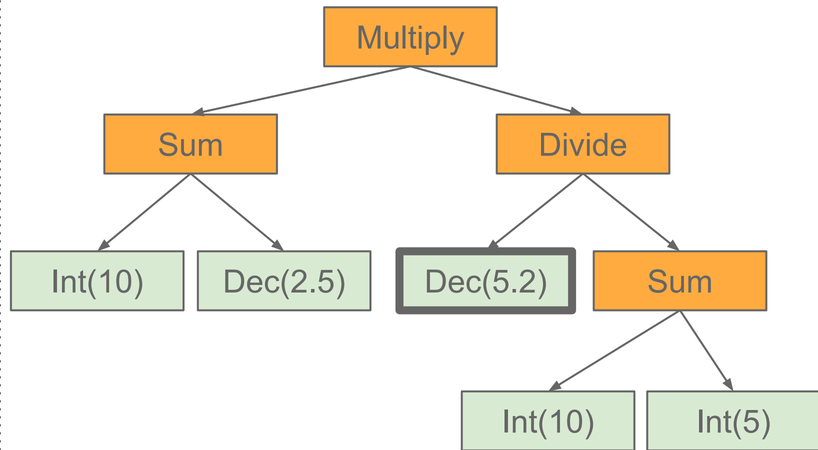

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)              => v.toDouble
  case DecValue(v)              => v
  case Sum(exp1, exp2)          =>
            evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
            evaluate(exp1) * evaluate(exp2)
  case Square(exp)              =>
            val v = evaluate(exp)
            v * v
  case Divide(exp1, exp2)    =>
            evaluate(exp1) / evaluate(exp2)
}
```



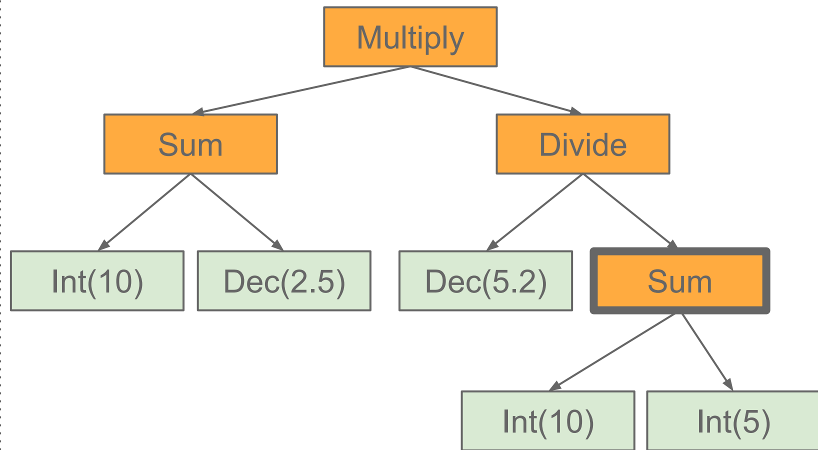

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)            => v.toDouble
  case DecValue(v)            => v
  case Sum(exp1, exp2)        =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)            =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)     =>
              evaluate(exp1) / evaluate(exp2)
}
```



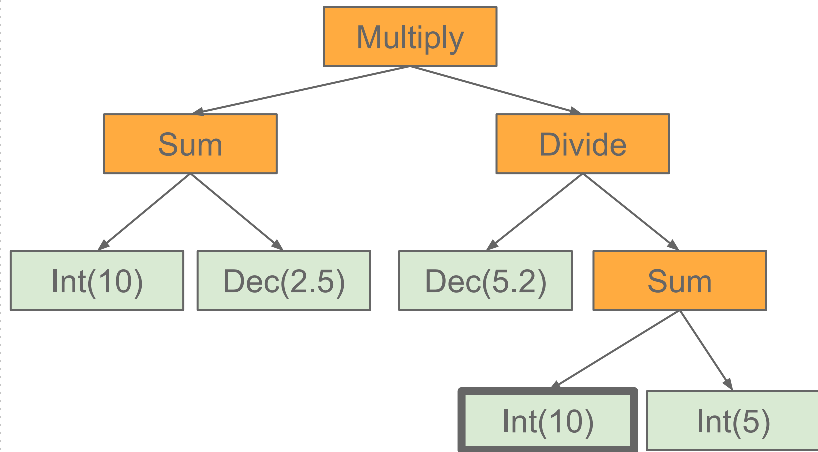

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```
val evaluate: Exp => Double = {
  case IntValue(v)              => v.toDouble
  case DecValue(v)              => v
  case Sum(exp1, exp2)          =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)              =>
               val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



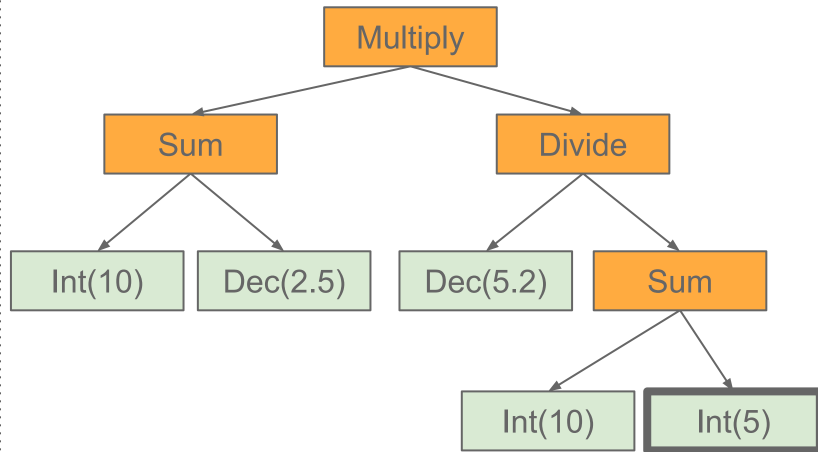

```
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```
val evaluate: Exp => Double = {
  case IntValue(v)            => v.toDouble
  case DecValue(v)            => v
  case Sum(exp1, exp2)        =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)            =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)     =>
              evaluate(exp1) / evaluate(exp2)
}
```



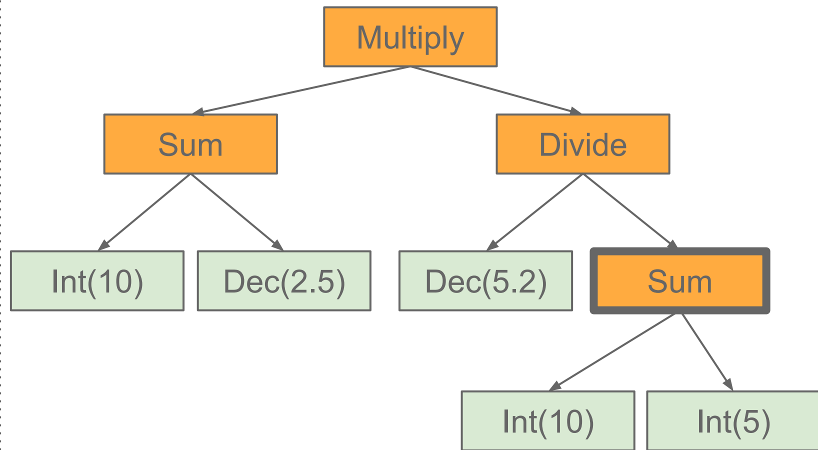

```
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)             => v.toDouble
  case DecValue(v)             => v
  case Sum(exp1, exp2)         =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)             =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)   =>
              evaluate(exp1) / evaluate(exp2)
}
```



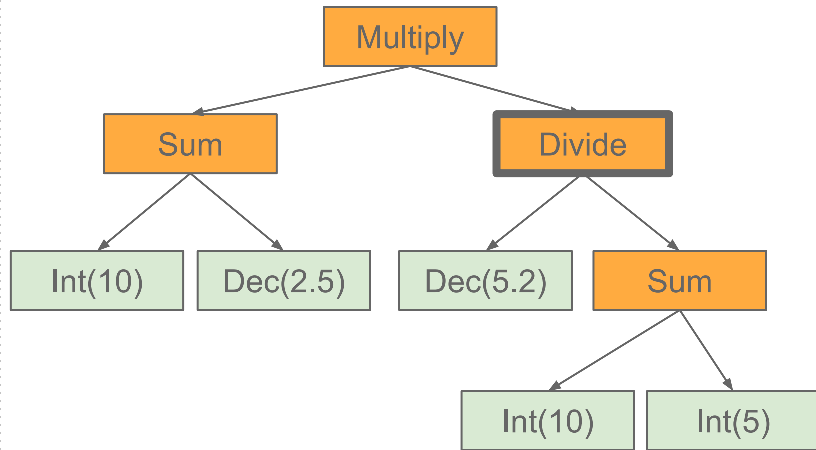

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
            evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
            evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
            val v = evaluate(exp)
            v * v
  case Divide(exp1, exp2)    =>
            evaluate(exp1) / evaluate(exp2)
}
```



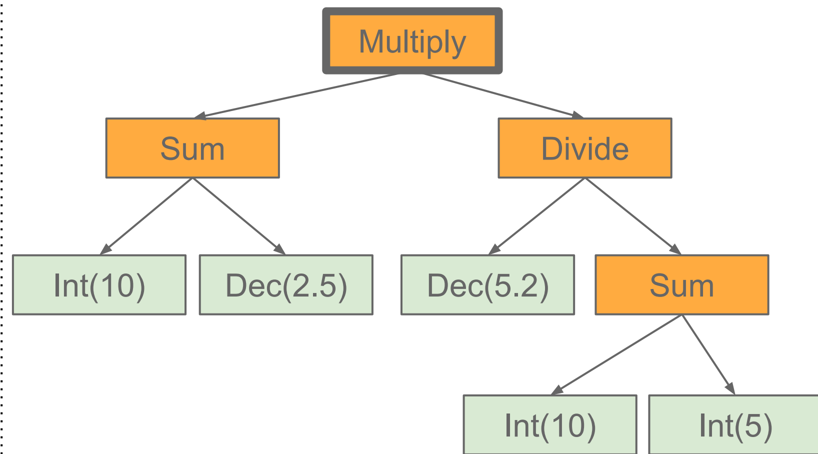

```
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)            => v.toDouble
  case DecValue(v)            => v
  case Sum(exp1, exp2)        =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)            =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```



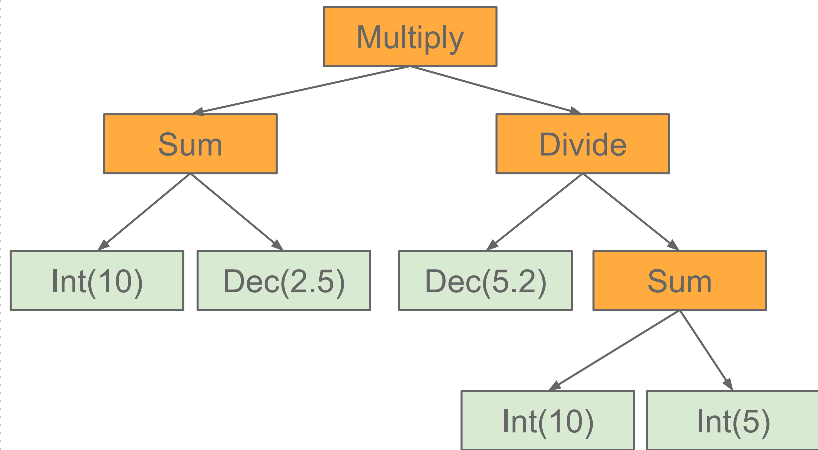

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
                evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
                evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
                 val v = evaluate(exp)
                v * v
  case Divide(exp1, exp2)    =>
                evaluate(exp1) / evaluate(exp2)
}
```



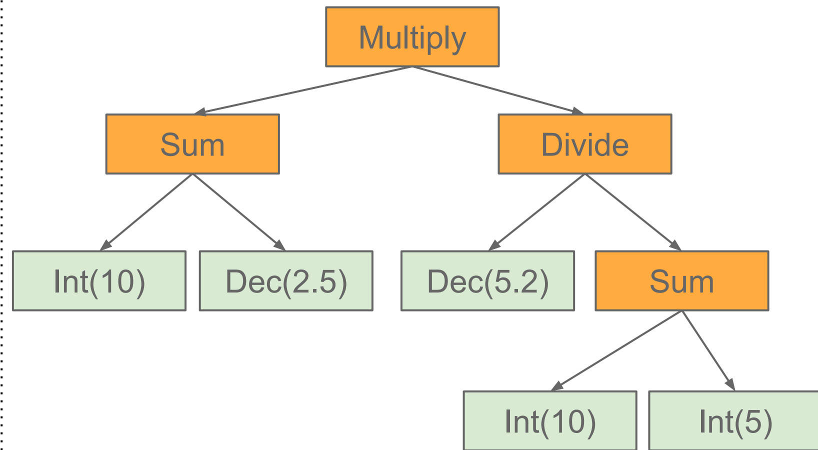

```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val mkString: Exp => String = {
  case IntValue(v)            => v.toString
  case DecValue(v)            => v.toString
  case Sum(exp1, exp2)        =>
       s"(${mkStr(exp1)} + ${mkStr(exp2)})"
  case Multiply(exp1, exp2) =>
       s"(${mkStr(exp1)} * ${mkStr(exp2)})"
  case Square(exp)            =>
                        s"(${mkStr(exp)})^2"
  case Divide(exp1, exp2)     =>
      s"( ${mkStr(exp1)} / ${mkStr(exp2)})"
}
```




```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
            evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
            evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
            val v = evaluate(exp)
            v * v
  case Divide(exp1, exp2)   =>
            evaluate(exp1) / evaluate(exp2)
}
```




```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[?](IntValue[?](10),
           IntValue[?](5))
          )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[?](IntValue[Unit](10),
           IntValue[Unit](5))
          )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[?](IntValue[Unit](10),
           IntValue[Unit](5))
          )
```

Exp[Unit]

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[?](IntValue[Unit](10),
           IntValue[Unit](5))
          )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[?](IntValue[Unit](10),
           IntValue[Unit](5))
          )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
```

```scala
val exp1: Exp[?] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
    )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[?] =
    Divide[?](
              DecValue[?](5.2),
              Sum[?](IntValue[?](10),
                     IntValue[?](5))
           )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[?] =
    Divide[?](
              DecValue[?](5.2),
              Sum[?](IntValue[Unit](10),
                     IntValue[Unit](5))
             )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[?] =
    Divide[?](
              DecValue[?](5.2),
              Sum[Exp[Unit]](IntValue[Unit](10),
                             IntValue[Unit](5))
             )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[?] =
    Divide[?](
              DecValue[Exp[Unit]](5.2),
              Sum[Exp[Unit]](IntValue[Unit](10),
                             IntValue[Unit](5))
              )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[?] =
    Divide[Exp[Exp[Unit]]](
             DecValue[Exp[Unit]](5.2),
             Sum[Exp[Unit]](IntValue[Unit](10),
                            IntValue[Unit](5))
             )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
              DecValue[Exp[Unit]](5.2),
              Sum[Exp[Unit]](IntValue[Unit](10),
                             IntValue[Unit](5))
              )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
               DecValue[Exp[Unit]](5.2),
               Sum[Exp[Unit]](IntValue[Unit](10),
                              IntValue[Unit](5))
               )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
             DecValue[Exp[Unit]](5.2),
             Sum[Exp[Unit]](IntValue[Unit](10),
                            IntValue[Unit](5))
             )
    )
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
val exp3: Exp = from(input)
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
                DecValue[Exp[Unit]](5.2),
                Sum[Exp[Unit]](IntValue[Unit](10),
                               IntValue[Unit](5))
             )
    )
val exp3: Exp[?] = from(input)
```

```scala
sealed trait Exp[A]
final case class IntValue[A](v: Int) extends Exp[A]
final case class DecValue[A](v: Double) extends Exp[A]
final case class Sum[A](exp1: A, exp2: A) extends Exp[A]
final case class Multiply[A](exp1: A, exp2: A) extends Exp[A]
final case class Divide[A](exp1: A, exp2: A) extends Exp[A]
final case class Square[A](exp: A) extends Exp[A]
```

```scala
val exp1: Exp =
  Sum(IntValue(10),
      IntValue(5))
     )
val exp2: Exp =
  Divide(
         DecValue(5.2),
         Sum(IntValue(10),
             IntValue(5))
         )
  )
val exp3: Exp = from(input)
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](IntValue[Unit](10),
                   IntValue[Unit](5))
                   )
val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
         DecValue[Exp[Unit]](5.2),
         Sum[Exp[Unit]](IntValue[Unit](10),
                        IntValue[Unit](5))
         )
    )
val exp3: Exp[Exp[Exp[Exp[Exp[Exp[Exp[Exp[Exp[Ex
```

Introducing: fix point data types!

Fix

I'M READY!
RUB THE BELLY!

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](
        IntValue[Unit](10),
        IntValue[Unit](5)
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](
        Fix(IntValue[Unit](10)),
        Fix(IntValue[Unit](5))
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Exp[Unit]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Exp[Exp[Unit]] =
    Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Exp[Exp[Unit]] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
        DecValue[Exp[Unit]](5.2),
        Sum[Exp[Unit]](
          IntValue[Unit](10),
          IntValue[Unit](5))
        )
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
        DecValue[Exp[Unit]](5.2),
        Sum[Exp[Unit]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        )
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
        DecValue[Exp[Unit]](5.2),
        Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        )
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
        Fix(DecValue[Fix[Exp]](5.2)),
        Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        )
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Divide[Exp[Exp[Unit]]](
        Fix(DecValue[Fix[Exp]](5.2)),
        Fix(Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        ))
    )
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Exp[Exp[Exp[Unit]]] =
    Fix(Divide[Fix[Exp]](
        Fix(DecValue[Fix[Exp]](5.2)),
        Fix(Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        ))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide[Fix[Exp]](
        Fix(DecValue[Fix[Exp]](5.2)),
        Fix(Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        ))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum[Fix[Exp]](
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide[Fix[Exp]](
        Fix(DecValue[Fix[Exp]](5.2)),
        Fix(Sum[Fix[Exp]](
          Fix(IntValue[Fix[Exp]](10)),
          Fix(IntValue[Fix[Exp]](5)))
        ))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide(
        Fix(DecValue(5.2)),
        Fix(Sum(
          Fix(IntValue(10)),
          Fix(IntValue(5))
        ))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide(
        Fix(DecValue(5.2)),
        Fix(Sum(
          Fix(IntValue(10)),
          Fix(IntValue(5))
        ))
    ))
```

```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide(
        Fix(DecValue(5.2)),
        Fix(Sum(
          Fix(IntValue(10)),
          Fix(IntValue(5))
        ))
    ))
```

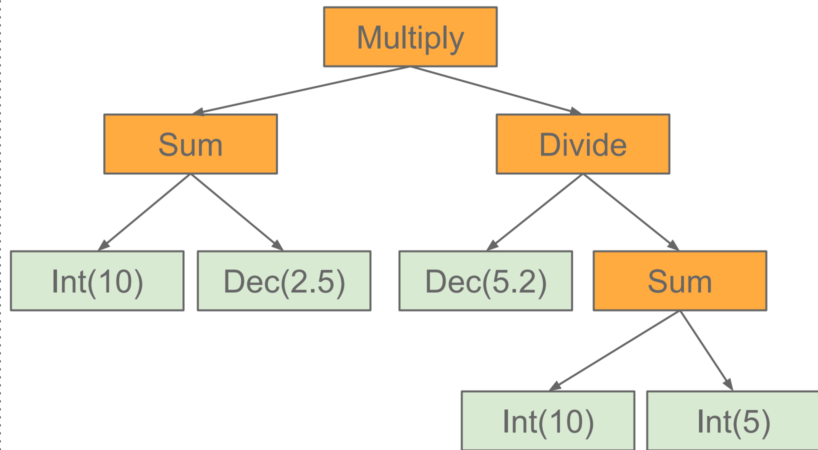
```scala
case class Fix[F[_]](unFix: F[Fix[F]])
```

```scala
val exp1: Fix[Exp] =
    Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
    ))

val exp2: Fix[Exp] =
    Fix(Divide(
        Fix(DecValue(5.2)),
        Fix(Sum(
          Fix(IntValue(10)),
          Fix(IntValue(5))
        ))
    ))

val exp3: Fix[Exp] = from(input)
```

```scala
val evaluate: Exp => Double = {
  case IntValue(v)           => v.toDouble
  case DecValue(v)           => v
  case Sum(exp1, exp2)       =>
              evaluate(exp1) + evaluate(exp2)
  case Multiply(exp1, exp2) =>
              evaluate(exp1) * evaluate(exp2)
  case Square(exp)           =>
              val v = evaluate(exp)
              v * v
  case Divide(exp1, exp2)    =>
              evaluate(exp1) / evaluate(exp2)
}
```




```scala
val exp = Multiply(
    Sum(IntValue(10),
        DecValue(2.5)),
    Divide(DecValue(5.2),
           Sum(IntValue(10),
               IntValue(5))
           )
)
```

Need something that will traverse the structure…

Catamorphism

greek κατα -“downwards” as in “catastrophe”

Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire

Erik Meijer [*] Maarten Fokkinga [†] Ross Paterson [‡]

Abstract

We develop a calculus for lazy functional programming based on recursion operators associated with data type definitions. For these operators we derive various algebraic laws that are useful in deriving and manipulating programs. We shall show that all

Missing Ingredients

1. Functor
2. Our Function
    a. a thing that is actually doing stuff

Functor 101

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}
```

```
def foo[A[_], T](a: A[T])
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_], T](a: A[T])(trans: T => Int)
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_], T](a: A[T])(trans: T => Int): A[Int] = ???
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_] : Functor, T](a: A[T])(trans: T => Int): A[Int] = ???
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_] : Functor, T](a: A[T])(trans: T => Int): A[Int] =  a.map(trans)
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_] : Functor, T](a: A[T])(trans: T => Int): A[Int] =  a.map(trans)




case class Container[T](t: T)

val r: Container[Int] =
          foo[Container, String](Container("bar"))(str => str.length)
```

Functor 101

```
trait Functor[F[_]] {
    def map[A, B](fa: F[A])(f: A => B): F[B]
}


def foo[A[_] : Functor, T](a: A[T])(trans: T => Int): A[Int] =  a.map(trans)

implicit val functor: Functor[Container] = new Functor[Container] {
   def map[A, B](c: Container[A])(f: A => B): Container[B] = Container(f(c.t))
}

case class Container[T](t: T)

val r: Container[Int] =
          foo[Container, String](Container("bar"))(str => str.length)
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
implicit val functor: Functor[Exp] = new Functor[Exp] {
    def map[A, B](exp: Exp[A])(f: A => B): Exp[B] =
```

```scala
sealed trait Exp
final case class IntValue(v: Int) extends Exp
final case class DecValue(v: Double) extends Exp
final case class Sum(exp1: Exp, exp2: Exp) extends Exp
final case class Multiply(exp1: Exp, exp2: Exp) extends Exp
final case class Divide(exp1: Exp, exp2: Exp) extends Exp
final case class Square(exp: Exp) extends Exp
```

```scala
implicit val functor: Functor[Exp] = new Functor[Exp] {
    def map[A, B](exp: Exp[A])(f: A => B): Exp[B] = exp match {
      case Sum(a1, a2) => Sum(f(a1), f(a2))
      case Multiply(a1, a2) => Multiply(f(a1), f(a2))
      case Divide(a1, a2) => Divide(f(a1), f(a2))
      case Square(a) => Square(f(a))
      case IntValue(v) => IntValue(v)
      case DecValue(v) => DecValue(v)
    }
  }
```

Missing Ingredients

1. Functor
2. Our Function

Missing Ingredients

1. ~~Functor~~
2. Our Function

Missing Ingredients

1. ~~Functor~~
2. F-Algebra

F-algebra

```
type Algebra[F[_], A] = F[A] => A
```

```scala
type Algebra[F[_], A] = F[A] => A
```

```scala
val evaluate: Algebra[Exp, Double] = { // Exp[Double] => Double
    case IntValue(v) => v.toDouble
    case DecValue(v) => v
    case Sum(a1, a2) => a1 + a2
    case Multiply(a1, a2) => a1 * a2
    case Square(a) => a * a
    case Divide(a1, a2) => a1 / a2
}
```

```scala
type Algebra[F[_], A] = F[A] => A
```

```scala
val evaluate: Algebra[Exp, Double] = { // Exp[Double] => Double
    case IntValue(v) => v.toDouble
    case DecValue(v) => v
    case Sum(a1, a2) => a1 + a2
    case Multiply(a1, a2) => a1 * a2
    case Square(a) => a * a
    case Divide(a1, a2) => a1 / a2
}

val mkStr: Algebra[Exp, String] = {
    case IntValue(v) => v.toString
    case DecValue(v) => v.toString
    case Sum(a1, a2) => s"($a1 + $a2)"
    case Multiply(a1, a2) => s"($a1 + $a2)"
    case Square(a) => s"($a)^2"
    case Divide(a1, a2) => s"($a1 + $a2)"
}
```

```scala
val evaluate: Algebra[Exp, Double] = { // Exp[Double] => Double
    case IntValue(v) => v.toDouble
    case DecValue(v) => v
    case Sum(a1, a2) => a1 + a2
    case Multiply(a1, a2) => a1 * a2
    case Square(a) => a * a
    case Divide(a1, a2) => a1 / a2
}
```

```scala
val evaluate: Algebra[Exp, Double] = { // Exp[Double] => Double
    case IntValue(v) => v.toDouble
    case DecValue(v) => v
    case Sum(a1, a2) => a1 + a2
    case Multiply(a1, a2) => a1 * a2
    case Square(a) => a * a
    case Divide(a1, a2) => a1 / a2
}

val exp2: Fix[Exp] = Fix(Divide(
      Fix(DecValue(5.2)),
      Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
      ))
  ))
```

```scala
val evaluate: Algebra[Exp, Double] = { // Exp[Double] => Double
    case IntValue(v) => v.toDouble
    case DecValue(v) => v
    case Sum(a1, a2) => a1 + a2
    case Multiply(a1, a2) => a1 * a2
    case Square(a) => a * a
    case Divide(a1, a2) => a1 / a2
}

val exp2: Fix[Exp] = Fix(Divide(
      Fix(DecValue(5.2)),
      Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
      ))
  ))

> exp2.cata(evaluate)
0.34666666666666667
```

```
val mkStr: Algebra[Exp, String] = {  // Exp[String] => String
    case IntValue(v) => v.toString
    case DecValue(v) => v.toString
    case Sum(a1, a2) => s"($a1 + $a2)"
    case Multiply(a1, a2) => s"($a1 + $a2)"
    case Square(a) => s"($a)^2"
    case Divide(a1, a2) => s"($a1 + $a2)"
}
```

```scala
val mkStr: Algebra[Exp, String] = {  // Exp[String] => String
    case IntValue(v) => v.toString
    case DecValue(v) => v.toString
    case Sum(a1, a2) => s"($a1 + $a2)"
    case Multiply(a1, a2) => s"($a1 + $a2)"
    case Square(a) => s"($a)^2"
    case Divide(a1, a2) => s"($a1 + $a2)"
}

val exp2: Fix[Exp] = Fix(Divide(
      Fix(DecValue(5.2)),
      Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
      ))
  ))
```

```scala
val mkStr: Algebra[Exp, String] = {  // Exp[String] => String
    case IntValue(v) => v.toString
    case DecValue(v) => v.toString
    case Sum(a1, a2) => s"($a1 + $a2)"
    case Multiply(a1, a2) => s"($a1 + $a2)"
    case Square(a) => s"($a)^2"
    case Divide(a1, a2) => s"($a1 + $a2)"
}

val exp2: Fix[Exp] = Fix(Divide(
      Fix(DecValue(5.2)),
      Fix(Sum(
        Fix(IntValue(10)),
        Fix(IntValue(5))
      ))
  ))

> exp2.cata(mkStr)
(5.2 + (10 + 5))
```

```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
            if(exp1 == exp2) => Square(optimize(exp1))
 case IntValue(v)            => IntValue(v)
 case DecValue(v)            => DecValue(v)
 case Sum(exp1, exp2)        => Sum(optimize(exp1), optimize(exp2))
 case Multiply(exp1, exp2)   => Multiply(optimize(exp1), optimize(exp2))
 case Square(exp)            => Square(optimize(exp))
 case Divide(exp1, exp2)     => Divide(optimize(exp1), optimize(exp2))
}
```

```scala
val optimize: Exp => Exp = {
 case Multiply(exp1, exp2)
            if(exp1 == exp2) => Square(optimize(exp1))
 case IntValue(v)               => IntValue(v)
 case DecValue(v)               => DecValue(v)
 case Sum(exp1, exp2)           => Sum(optimize(exp1), optimize(exp2))
 case Multiply(exp1, exp2)      => Multiply(optimize(exp1), optimize(exp2))
 case Square(exp)               => Square(optimize(exp))
 case Divide(exp1, exp2)        => Divide(optimize(exp1), optimize(exp2))
}
```

```scala
val optimize: Algebra[Exp, Fix[Exp]] = { // Exp[Fix[Exp]] => Fix[Exp]
   case Multiply(Fix(a1), Fix(a2)) if(a1 == a2) => Fix(Square(Fix(a1)))
   case other => Fix(other)
}
```

```scala
val optimize: Algebra[Exp, Fix[Exp]] = { // Exp[Fix[Exp]] => Fix[Exp]
  case Multiply(Fix(a1), Fix(a2)) if(a1 == a2) => Fix(Square(Fix(a1)))
  case other => Fix(other)
}
```

```scala
val optimize: Algebra[Exp, Fix[Exp]] = { // Exp[Fix[Exp]] => Fix[Exp]
  case Multiply(Fix(a1), Fix(a2)) if(a1 == a2) => Fix(Square(Fix(a1)))
  case other => Fix(other)
}

val aTimesAExp: Fix[Exp] =
    Fix(Multiply(
      Fix(Sum(
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](20))
      )),
      Fix(Sum(
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](20))
      ))
  ))
```

```scala
val optimize: Algebra[Exp, Fix[Exp]] = { // Exp[Fix[Exp]] => Fix[Exp]
   case Multiply(Fix(a1), Fix(a2)) if(a1 == a2) => Fix(Square(Fix(a1)))
   case other => Fix(other)
}

val aTimesAExp: Fix[Exp] =
    Fix(Multiply(
      Fix(Sum(
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](20))
      )),
      Fix(Sum(
        Fix(IntValue[Fix[Exp]](10)),
        Fix(IntValue[Fix[Exp]](20))
      ))
   ))

> aTimesAExp.cata(optimize)

Fix(Square(Fix(Sum(Fix(IntValue(10)),Fix(IntValue(20))))))
```

zoo of morphisims

anamorphism & hylomorphism

Anamorphism

- Constructs a structure from a value

Anamorphism

- Constructs a structure from a value
- Co- part of catamorphism

Anamorphism

- Constructs a structure from a value
- Co- part of catamorphism
- Instead of using Algebra[F[_], A] we will use Coalgebra[F[_], A]

Anamorphism

- Constructs a structure from a value
- Co- part of catamorphism
- Instead of using Algebra[F[_], A] we will use Coalgebra[F[_], A]

Example: given Int, construct expression called divisors that

- is multiplication of series of 2s & one odd value
- once evaluated will return initial Int

e.g 28 = 2 * 2 * 7

```scala
type Coalgebra[F[_], A] = A => F[A]
```

```scala
type Coalgebra[F[_], A] = A => F[A]
```

```scala
val divisors: Coalgebra[Exp, Int] = {
    case n if(n % 2 == 0 && n != 2) => Multiply(2, n / 2)
    case n => IntValue(n)
}
```

```
type Coalgebra[F[_], A] = A => F[A]
```

```
val divisors: Coalgebra[Exp, Int] = {
    case n if(n % 2 == 0 && n != 2) => Multiply(2, n / 2)
    case n => IntValue(n)
}


> 12.ana[Fix, Exp](divisors)
```

```scala
type Coalgebra[F[_], A] = A => F[A]
```

```scala
val divisors: Coalgebra[Exp, Int] = {
    case n if(n % 2 == 0 && n != 2) => Multiply(2, n / 2)
    case n => IntValue(n)
}


> 12.ana[Fix, Exp](divisors)

Fix(Multiply(Fix(IntValue(2)),Fix(Multiply(Fix(IntValue(2)),Fix(Multiply(Fix(IntVal
ue(2)),Fix(IntValue(7))))))))
```

Hylomorphism

Hylomorphism

- Constructs and then deconstructs a structure from a value

Hylomorphism

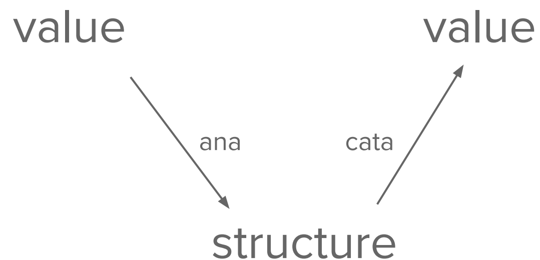
- Constructs and then deconstructs a structure from a value
- Anamorphism followed by catamorphism

Hylomorphism

- Constructs and then deconstructs a structure from a value
- Anamorphism followed by catamorphism
- Difference: evaluated in a single pass

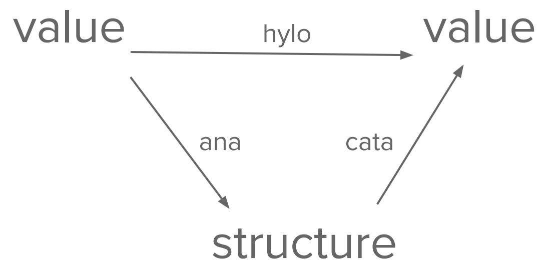
Hylomorphism

- Constructs and then deconstructs a structure from a value
- Anamorphism followed by catamorphism
- Difference: evaluated in a single pass



Hylomorphism

- Constructs and then deconstructs a structure from a value
- Anamorphism followed by catamorphism
- Difference: evaluated in a single pass



```scala
val divisors: Coalgebra[Exp, Int] = { … }  // Double => Exp[Double]
val evaluate: Algebra[Exp, Double] = { … } // Exp[Double] => Double
```

```scala
val divisors: Coalgebra[Exp, Int] = { … }  // Double => Exp[Double]
val evaluate: Algebra[Exp, Double] = { … } // Exp[Double] => Double


describe("divisors") {
  it("once evaluated will give initial value") {
    forAll(positiveInt) { n =>
      n.ana(divisiors).cata(evaluate) shouldEqual(n)
    }
}
```

```scala
val divisors: Coalgebra[Exp, Int] = { … }  // Double => Exp[Double]
val evaluate: Algebra[Exp, Double] = { … } // Exp[Double] => Double


describe("divisors") {
  it("once evaluated will give initial value") {
    forAll(positiveInt) { n =>
      n.hylo(evaluate, divisors) shouldEqual(n)
    }
}
```

What else?

To sum it up

- Recursion is hard, but it is ubiquitous
- Explicit recursion is tedious & error-prone
- Recursion schemas
  - set of composable combinators
  - well defined in both academia & industry
- Fix point data types
- You don’t have to do FP to take advantage of recursion schemes
- You can be famous! Build stuff!

References

[NATO68] - http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

[MTEU16] - https://www.infoq.com/presentations/engineer-practices-techniques

[BdM97] - R. Bird and O. de Moor. Algebra of Programming. Series in Computer Science. Prentice-Hall International, 1997. C.A.R. Hoare, series editor.

[MFP91] - http://eprints.eemcs.utwente.nl/7281/01/db-utwente-40501F46.pdf

[STOW14] - http://blog.sumtypeofway.com/an-introduction-to-recursion-schemes/

Pawel Szulc

Pawel Szulc

@rabbitonweb

Pawel Szulc

@rabbitonweb
paul.szulc@gmail.com

Pawel Szulc

@rabbitonweb
paul.szulc@gmail.com
http://rabbitonweb.com