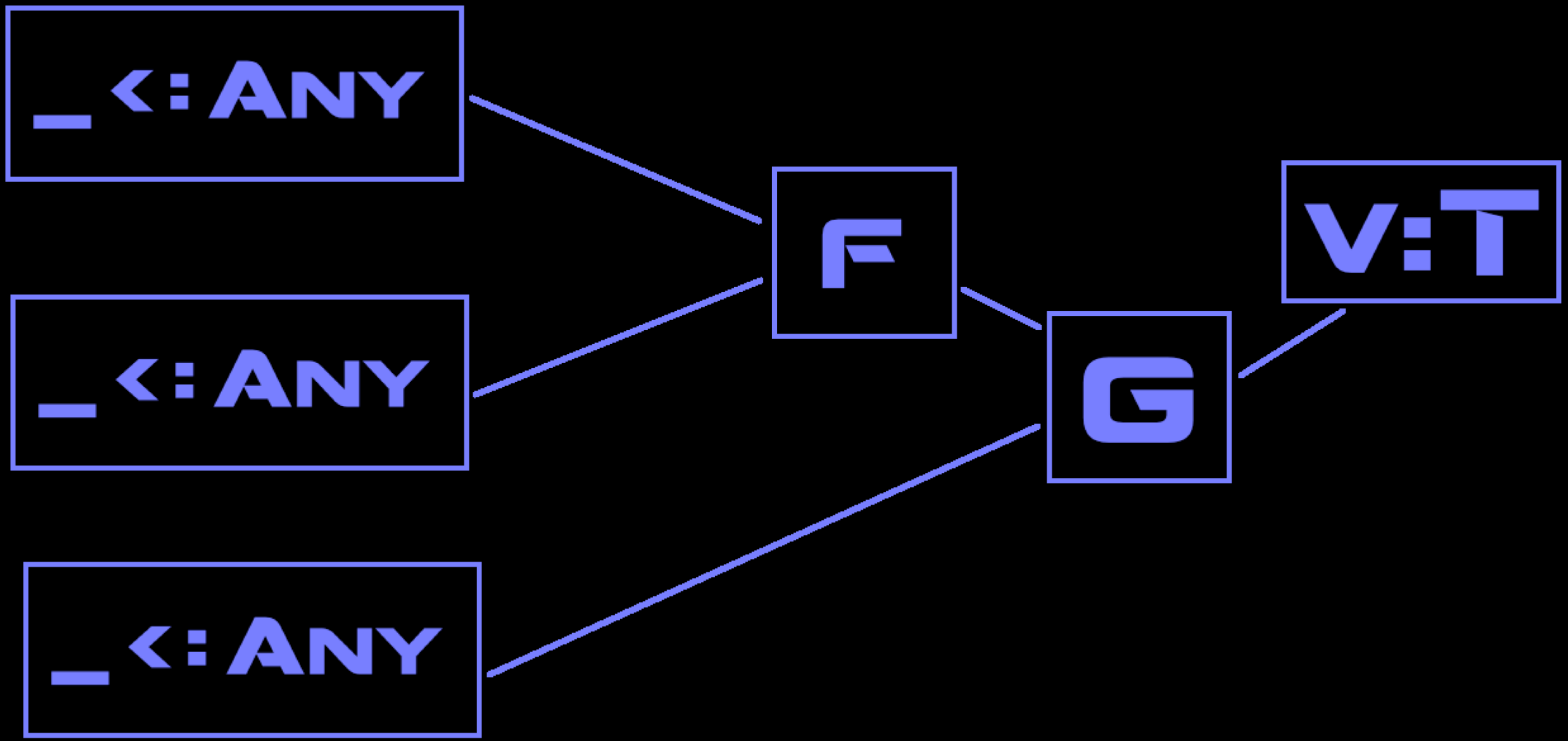
Purely Functional Semantic and Syntax Expression Composition



_<:Any
_<:Any
_<:Any
F
G
V:T

Compose Expressions, impose type information and handle errors

$$\Big(length(\text{''}\ Shostakovich\ \text{''}) + 5\Big) < 47$$

$$\implies True$$

$$\Big(length\big(KobayashiMaru\big) + KHAAANNNN!\Big) < 47 \implies \{KobayashiMaru, KHAAANNNN!\}$$

$$(1/0) * 8 \implies \{DivideByZero\}$$

Sample composed expression w/ evaluation

```scala
val expr = ('a & 'b.str.reverse) & 'c.str.sub(3, 6)

val weaklyTypedSource : Map[String, Any] = Map(
  "a" -> "Evelyn",
  "b" -> "ecirtaeB",
  "c" -> "SPOHallfIi"
)

val result = expr(weaklyTypedSource)
```

Monads

Kleisli

Validated
(Applicative Functors)

Spire

cats

Shapeless

$A \Rightarrow M[B]$ lifted into Kleisli[M, A, B]

```scala
// The Reader Monad is a special-case of Kleisli
type Reader[A, B] = Kleisli[Id, A, B]

val length = Kleisli[Option, String, Int] { s => Monad[Option] pure s.length }
val even = Kleisli[Option, Int, Boolean] { i => Some(i).map(_ % 2 == 0) }

// Composition works because when we have a FlatMap[Option] in scope
scala> (length andThen even).run("the length is even") exists identity
res3: Boolean = true
```

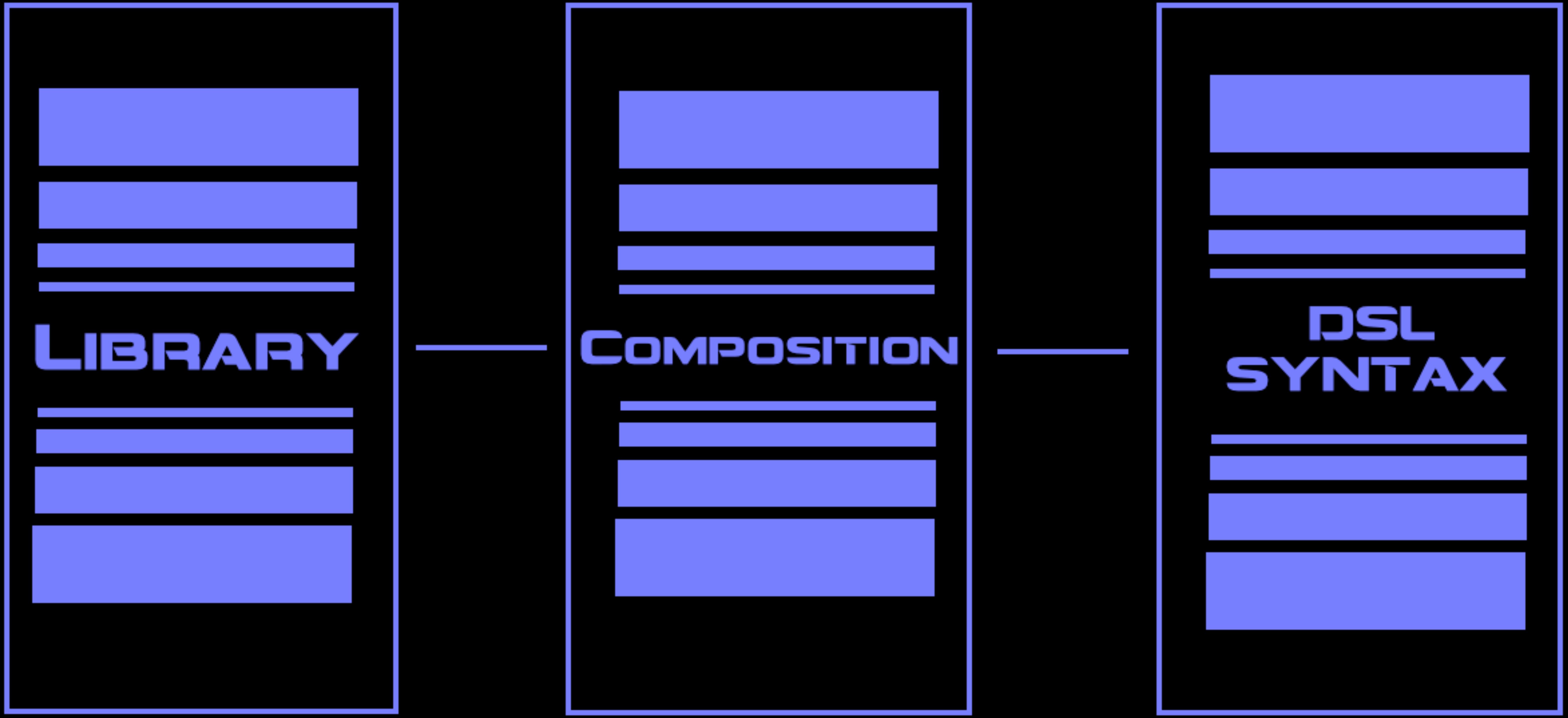
composition via for-comprehension

```
val both =
  for {
    len <- Kleisli[Option, String, Int] { s => Some(s.length) }
    rev <- Kleisli[Option, String, String] { s => Some(s.reverse) }
  } yield (length, reverse)

scala> both.run("Solzhenitsyn")
res17: Option[(Int, String)] = Some((12,nystinehzloS))
```

Injecting Enterprise in your Microservice?

```scala
trait Store[M[_], A, B] {

  def loader : Kleisli[M, A, B]
  def writer : Kleisli[M, (A, B), Unit]

}

// Assuming we want to use scalaz.Task and have some types defined (Profile, Metadata,
// and StoredObject) where Profile <: StoredObject and Metadata <: StoredObject
val sampleStore : Store[scalaz.Task, Symbol, StoredObject]

val task : Task[(Profile, Metadata)] = for {
  profile <- sampleStore.loader.run('profile)
  metadata <- sampleStore.loader.run('metadata)
  _ <- sampleStore.writer.write('profile -> profile.refresh)
  _ <- sampleStore.writer.write('context -> "default")
} yield (profile, metadata)

task.runAsync(handler) // Run the combined task asyncronously
```

The Trek Begins ...


LIBRARY
COMPOSITION
DSL SYNTAX

Core Definitions

```
case class EvalFault(fault: Symbol, message: String)

type Term[A] = ValidatedNel[EvalFault, A]

type Expr[A, B] = Kleisli[Term, A, B]
type In[A] = Expr[Any, A]  // Edge of the world
```

Follow the Types

Values of `In[A]` are "mapped" to values of `Expr[Source, Result]`

Our "Library" of operations

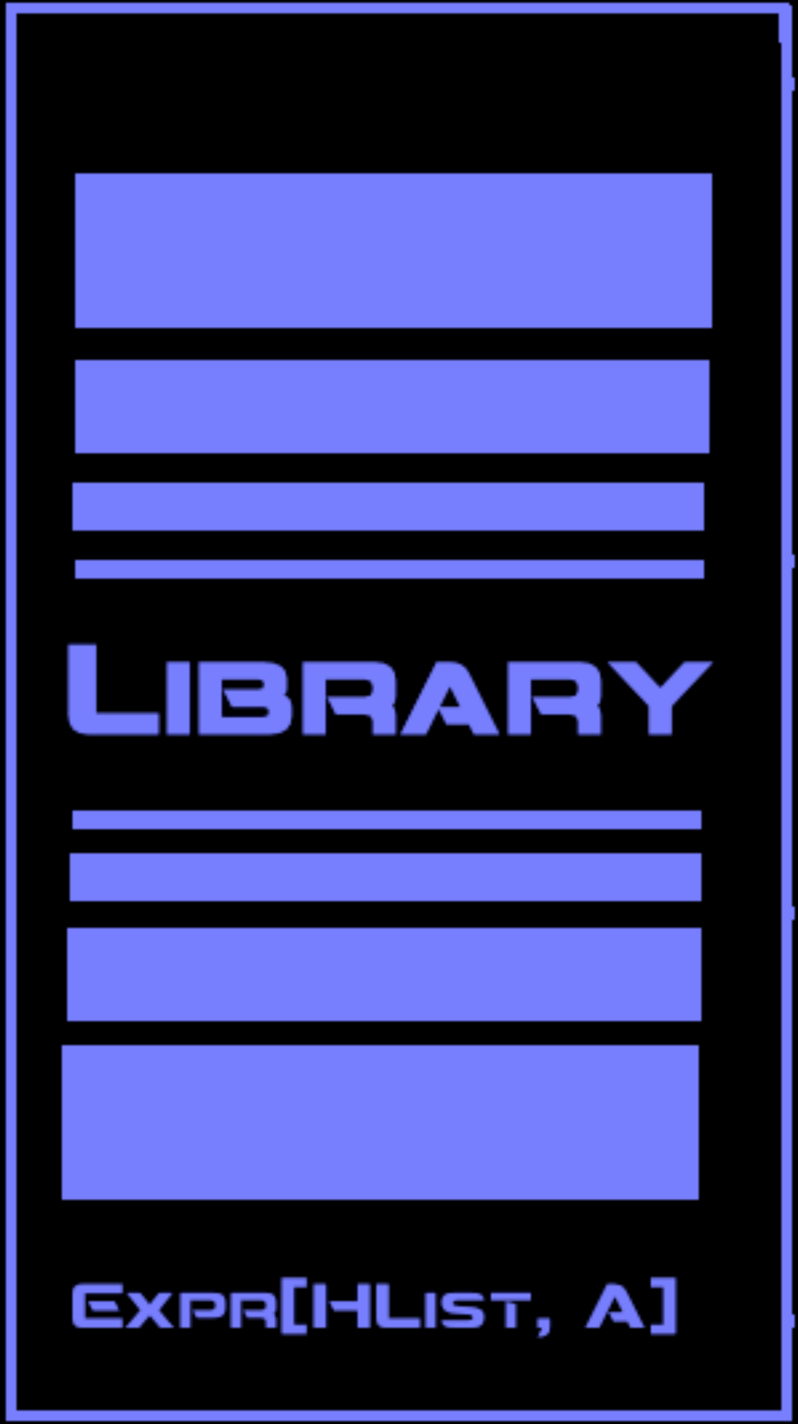
```
object Library {

  val concat: Expr[String :: String :: HNil, String] =
    lift2 { case l :: r :: HNil => (l |@| r).map {_ + _}}

  val reverse: Expr[String :: HNil, String] =
    lift1 { case s :: HNil => s.map(_.reverse) }

  // Continued ...
```

```scala
def add[A: Numeric]: Expr[A :: A :: HNil, A] =
  lift2 { case l :: r :: HNil => (l |@| r).map {_ + _} }
def minus[A: Numeric]: Expr[A :: A :: HNil, A] =
  lift2 { case l :: r :: HNil => (l |@| r).map { _ - _ } }
def mult[A: Numeric]: Expr[A :: A :: HNil, A] =
  lift2 { case l :: r :: HNil => (l |@| r).map { _ * _ } }

// Continued ...
```

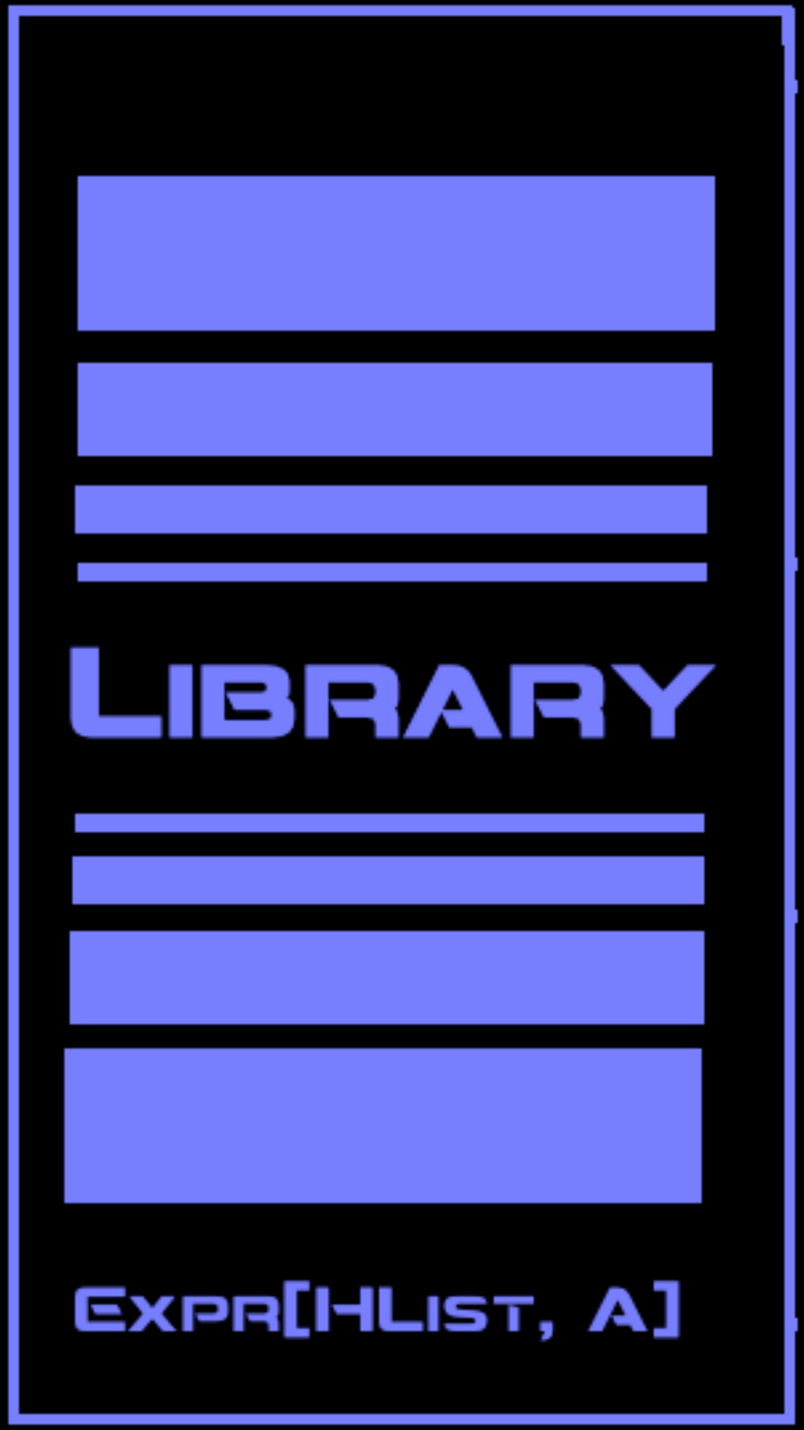
```scala
  def div[A: Numeric]: Expr[A :: A :: HNil, A] =
    composeLift2[A, A, A] {
      case l :: Valid(r) :: HNil if r == 0 =>
        l :: EvalFault('divByZero, "").invalidNel :: HNil
      case l :: r :: HNil =>
        l :: r :: HNil
    }
    { case l :: r :: HNil => (l |@| r).map { _ / _ } }

  def negate[A: Numeric]: Expr[A :: HNil, A] =
    lift1 { case a :: HNil => a.map(-_) }
} // End of object Library
```

So ...

What's the deal with these `liftN` and `composeLiftN` functions?



$$
\begin{tikzcd}
\texttt{f: (Term[O] :: HNil) => Term[A])} \arrow[d, "\texttt{lift1}"] \\
\texttt{Expr[O :: HNil, A]}
\end{tikzcd}
$$

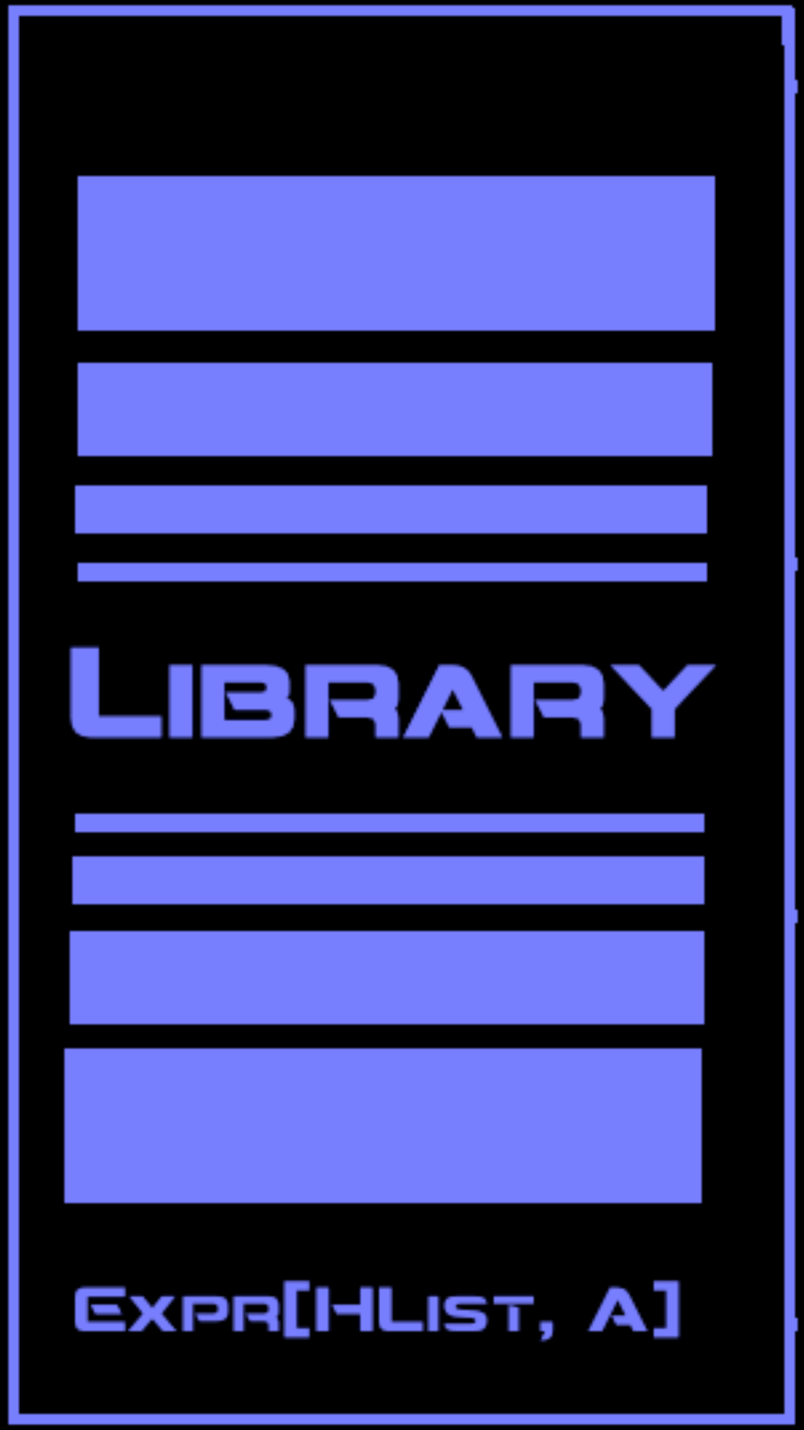


f: (Term[L] :: Term[R] :: HNil)
=> Term[A])

↓ lift2

Expr[L :: R :: HNil, A]

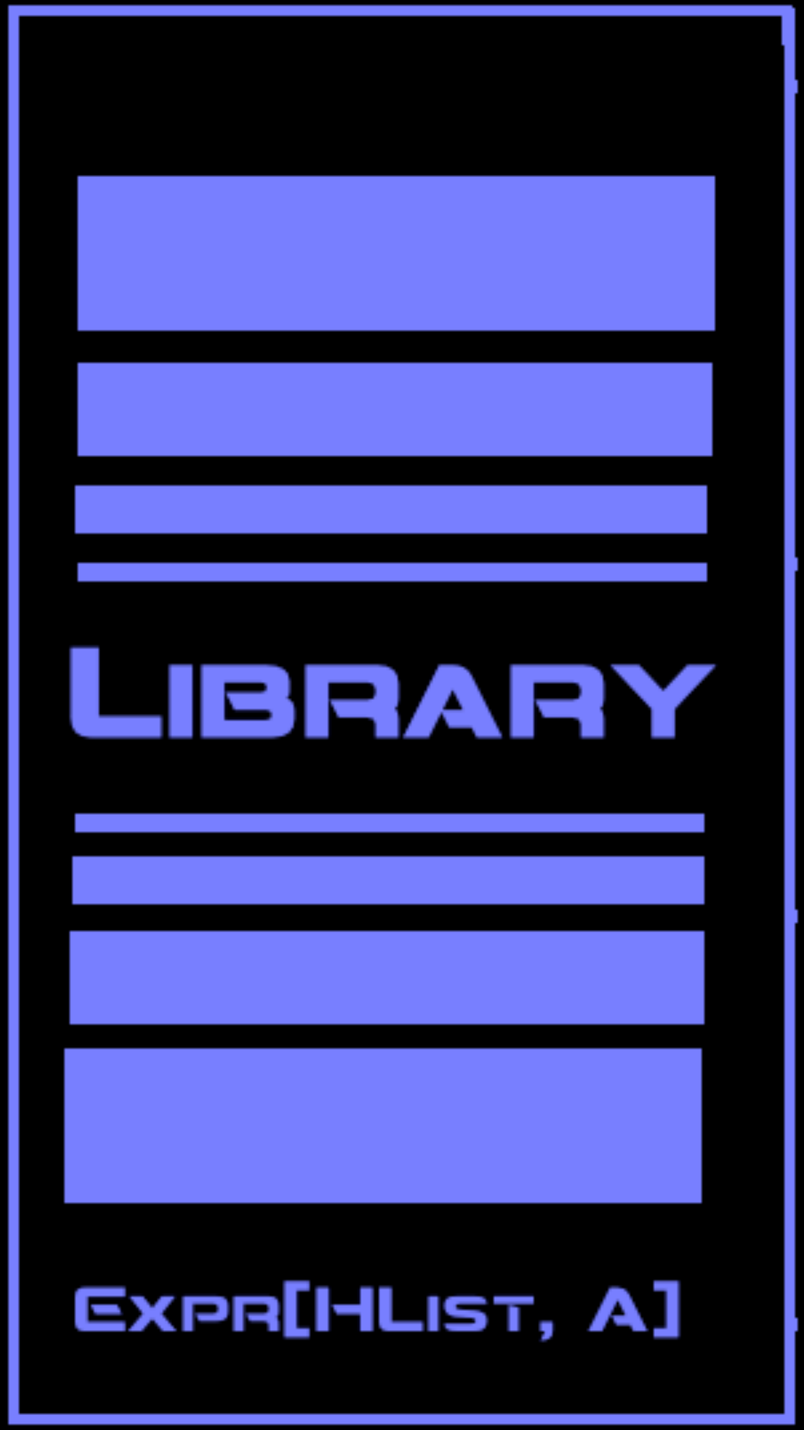
(f: (Term[O] :: HNil) => (Term[O] :: HNil))
(g: (Term[O] :: HNil) => Term[A])



$\downarrow$ ComposeLift1

Expr[O :: HNil, A]

(f: (Term[L] :: Term[R] :: HNil) => (Term[L] :: Term[R] :: HNil))
(g: (Term[L] :: Term[R] :: HNil) => Term[A])



ComposeLift2

Expr[L :: R :: HNil, A]

```scala
def lift1[O, A]
  (f: (Term[O] :: HNil) => Term[A]) : Expr[O :: HNil, A] =
    Kleisli[Term, O :: HNil, A] { case op :: HNil =>
      f(Applicative[Term].pure(op) :: HNil)
    }



def lift2[L, R, A]
  (f: (Term[L] :: Term[R] :: HNil) => Term[A]) : Expr[L :: R :: HNil, A] =
    Kleisli[Term, L :: R :: HNil, A] { case l :: r :: HNil =>
      f(Applicative[Term].pure(l) :: Applicative[Term].pure(r) :: HNil)
    }
```

```scala
def composeLift1[O, A]
  (f: (Term[O] :: HNil) => (Term[O] :: HNil))
  (g: (Term[O] :: HNil) => Term[A]) : Expr[O :: HNil, A] =
    lift1(f andThen g)



def composeLift2[L, R, A]
  (f: (Term[L] :: Term[R] :: HNil) => (Term[L] :: Term[R] :: HNil))
  (g: (Term[L] :: Term[R] :: HNil) => Term[A]) : Expr[L :: R :: HNil, A] =
    lift2(f andThen g)
```

Follow The Types

```
{ case s :: HNil => s.map(_.length) }
```

$

```
lift1[String, Int]
  (f: (Term[String] :: HNil) => Term[Int])
  : Expr[String :: HNil, Int]
```

```
len: Expr[String :: HNil, Int]
```

#

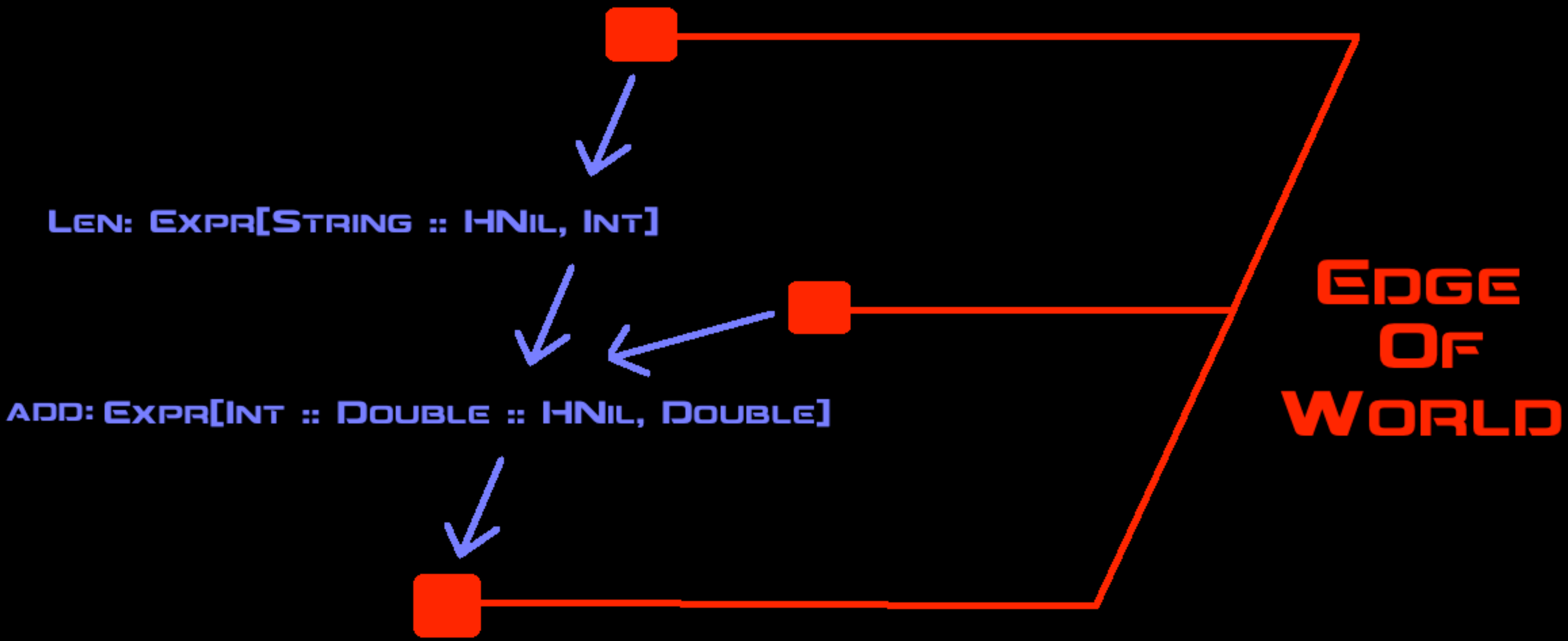
```
{ case l :: r :: HNil =>
    (l |@| r).map {_ + _} }
```

```
add: Expr[Int :: Double :: HNil, Double]
```

```
lift2[Int, Double, Double]
  (f: (Term[Int] :: Term[Double] :: HNil) =>
   Term[Double])
  : Expr[Int :: Double :: HNil, Double]
```

#

Follow The Types



```scala
object MapAdapter {
  type Key = String
  type Source = Map[Key, Any] // Weakly-typed Source

  implicit val strIn: In[String] = // Remember: type In[A] = Expr[Any, A]
    Kleisli[Term, Any, String](_.toString.validNel)

  implicit val doubleIn : In[Double] = Kleisli[Term, Any, Double] { v =>
    Xor.catchOnly[NumberFormatException](v.toString.toDouble)
      .leftMap(th => NonEmptyList[EvalFault](EvalFault('expectedDouble, th.getMessage))).toValidated
  }

  implicit val intIn : In[Int] = Kleisli[Term, Any, Int] { v =>
    Xor.catchOnly[NumberFormatException](v.toString.toInt)
      .leftMap(th => NonEmptyList[EvalFault](EvalFault('expectedInt, th.getMessage))).toValidated
  }

  def read[A](k: Key)(implicit i: In[A]): Expr[Source, A] =
    Kleisli[Term, Source, A] {
      _.get(k).map(i.run) getOrElse EvalFault('keyMissing, k).invalidNel
    }
}
```

```scala
object Syntax {
  import MapAdapter._
  import Library._

  implicit def doubleReader1(symbol: Symbol) : NumSymbolExpr[Double] = read[Double](symbol.name)
  implicit def doubleReader2(symbol: Symbol) : Expr[Source, Double] = read[Double](symbol.name)

  def int[A: Numeric](expr: Expr[Source, A]) : Expr[Source, Int] = expr.map(_.toInt)
  def double[A: Numeric](expr: Expr[Source, A]) : Expr[Source, Double] = expr.map(_.toDouble)
  def str[A <: Any](expr: Expr[Source, A]) : Expr[Source, String] = expr.map(_.toString)

  implicit def strReader1(symbol: Symbol) : StrSymbolExpr = read[String](symbol.name)
  implicit def strReader2(symbol: Symbol) : Expr[Source, String] = read[String](symbol.name)

  implicit def const[A](k: A) : Expr[Source, A] = Kleisli[Term, Source, A] { _ =>
    k.validNel
  }
// Continued ...
```

```scala
  implicit class SymbolOps(val symbol: Symbol) extends AnyVal {
    def int : Expr[Source, Int] = read[Int](symbol.name)
    def double : Expr[Source, Double] = read[Double](symbol.name)
    def str : Expr[Source, String] = read[String](symbol.name)
  }

  implicit class StrSymbolExpr(val l: Expr[Source, String]) extends AnyVal {
    def &(r: Expr[Source, String]) : Expr[Source,String] = eval2(l, r, concat)
    def reverse : Expr[Source, String] = eval1(l, )
  }

  implicit class NumSymbolExpr[A: Numeric](l: Expr[Source, A]) {
    def +(r: Expr[Source, A]) : Expr[Source, A] = eval2(l, r, add)
    def -(r: Expr[Source, A]) : Expr[Source, A] = eval2(l, r, minus)
    def *(r: Expr[Source, A]) : Expr[Source, A] = eval2(l, r, mult)
    def /(r: Expr[Source, A]) : Expr[Source, A] = eval2(l, r, div)
    def neg: Expr[Source, A] = eval1(l, negate)
  }
} // End of object Syntax
```

```scala
def eval1[A, B, C](operand: Expr[A, B], op: Expr[B :: HNil, C]) : Expr[A, C] =
  Kleisli[Term, A, C] { source => join1(operand :: HNil).run(source) match {
    case i @ Invalid(_) => i
    case Valid(v :: HNil) => op.run(v :: HNil)
  }}

def eval2[A, B, C, D](l: Expr[A, B], r: Expr[A, C], op: Expr[B :: C :: HNil, D]) : Expr[A, D] =
  Kleisli[Term, A, D] { source => join2(l :: r :: HNil).run(source) match {
    case i @ Invalid(_) => i
    case Valid(lv :: rv :: HNil) => op.run(lv :: rv :: HNil)
  }}

def join1[A, O](operand: Expr[A, O] :: HNil) : Expr[A, O :: HNil] =
  Kleisli[Term, A, O :: HNil] { a => operand match {
    case op :: HNil => op.run(a).map(_ :: HNil)
  }
}

def join2[A, L, R](operands: Expr[A, L] :: Expr[A, R] :: HNil) : Expr[A, L :: R :: HNil] =
  Kleisli[Term, A, L :: R :: HNil] { a => operands match {
    case l :: r :: HNil => (l.run(a) |@| r.run(a)).map(_ :: _ :: HNil)
  }
}
```

Follow The Types

```
read[String](k: Key)(implicit i: In[String] = inStr): Expr[Source, String]

eval1[Source, String, Int](operand: Expr[Source, String],
  op: Expr[String :: HNil, Int]) : Expr[Source, Int]
```

Fully Composed
Expression

```
Len: Expr[String :: HNil, Int]
```

And Now, this works!

```scala
val expr = ('a & 'b.str.reverse) & 'c.str.sub(3, 6)

val weaklyTypedSource : Map[String, Any] = Map(
  "a" -> "Evelyn",
  "b" -> "ecirtaeB",
  "c" -> "SPOHallfIi"
)

val result = expr(weaklyTypedSource)
```

Thank you

Questions?

Source:

https://github.com/ryanonsrc/catfarm/blob/master/src/main/scala/io/nary/catfarm/xdsl/DSL.scala

This code is solely for instructional/demonstration purposes and shouldn't be used for production. It is also likely that this code may be changed and/or moved